Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Andrej Podzimek

Read-Copy-Update for OpenSolaris

Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký

Degree program: Computer Science, Software Systems

2010

I would like to thank the thesis supervisor, Martin Děcký, for his valuable advice on both the text and the accompanying software project implementation.

Thesis Title: **Read-Copy-Update for OpenSolaris**
Author: Andrej Podzimek
Department: Department of Distributed and Dependable Systems
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail: `decky@d3s.mff.cuni.cz`

Abstract:

The goal of this thesis is to design and implement a RCU (Read-Copy-Update) synchronization mechanism for OpenSolaris.

The main purpose of the RCU mechanism is to increase concurrency in readers-writers synchronization scenarios, especially in SMP (Symmetric Multiprocessing) environments. This improvement is achieved by keeping multiple versions of the protected data which enables readers and writers to run in parallel. The RCU synchronization has already been implemented multiple times and is used in the Linux kernel.

The thesis includes analysis of existing RCU implementations, possible benefits of RCU in the ONNV (OpenSolaris) kernel and a prototype implementation in ONNV. It also suggests possible applications of RCU where lockless synchronization is already applied. The prototype implementation is compared with its counterpart in the Linux kernel.

Keywords: RCU Read-Copy-Update OpenSolaris kernel UTS synchronization

Název práce: **Read-Copy-Update pro OpenSolaris**
Autor: Andrej Podzimek
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí práce: Mgr. Martin Děcký
e-mail vedoucího: `decky@d3s.mff.cuni.cz`

Abstrakt:

Cílem práce je návrh a implementace mechanismu RCU (Read-Copy-Update) pro OpenSolaris.

Hlavním účelem mechanismu RCU je zvýšení souběžnosti (paralelismu) při synchronizaci mezi čtenáři a zapisovateli, zejména u víceprocesorových systémů. Tohoto zlepšení se dosáhne udržováním několika verzí chráněných dat, což umožňuje čtenářům i zapisovatelům pracovat souběžně. Synchronizace typu RCU už byla několikrát implementována a používá se v kernelu Linux.

Tato práce zahrnuje analýzu stávajících implementací RCU, možných výhod RCU pro kernel ONNV (OpenSolaris) a prototypovou implementaci pro ONNV. Zároveň navrhuje možnosti využití RCU v místech, kde se již používá neblokující sychronizace. Srovnává prototypovou implementaci s odpovídající částí kernelu Linux.

Klíčová slova: RCU Read-Copy-Update OpenSolaris kenel UTS synchronizace

Contents

List of Figures

Listings

Chapter 1

Introduction to RCU

This chapter gives a description of the RCU synchronization mechanism. It also mentions the main advantages and disadvantages of RCU when compared to synchronization based on mutual exclusion.

1.1 What Is RCU (and What It Is Not)

RCU is a means of waiting for procedures or computations (such as loads from and stores to a certain area of memory) to finish. [11] It is not a synchronization primitive based on mutual exclusion. In fact avoiding mutual exclusion is its main purpose. It is a framework useful for designing parallel algorithms. The RCU API defines three entities: *readers*, *writers* and *reclaimers*.

1.1.1 RCU Writers

Writers are entities that modify a data structure. For example, they add new items to linked lists or hash tables. Using the RCU API, writers can run in parallel with readers, as long as they follow certain constraints. First, the RCU API does not provide any means of synchronization among writers. Consequently, writers must use other synchronization primitives (based either on mutual exclusion or on non-blocking synchronization) to make sure the data structure is in a consistent state after their modifications. Second, concurrently running readers must always see a consistent state of the data structure. This implies that all assignments made by writers **and** visible to readers must be atomic. [12]

The second constraint mentioned above has important implications. Only word-sized assignments are atomic on current processors. This is why data structures protected by RCU mostly use the Read-Copy-Update data manipulation, which gave RCU its name. For example, when the writer needs to make (non-atomic) changes to a data structure accessible through a pointer, it must make sure the changes appear to occur atomically, as observed by readers. This is achieved by copying the whole data structure to a location out of reach of the readers. The copy can be safely modified, since only the writer can access it. When all the modifications are complete (and globally visible), the writer can replace the pointer to the original data structure with a pointer to the copy. Word-aligned pointer assignments are atomic, which guarantees that readers will always see a consistent state of the data structure.

1.1.2 RCU Readers

Readers can be perceived as threads accessing data without modifying its structure. They **never** need to block when entering their critical sections. They are guaranteed to see a consistent state of the data structure, provided that they follow certain rules when reading the data. For example, a reader iterating over a doubly linked list protected by RCU is not allowed to change the direction of iteration. Most importantly, readers can run in parallel with writers. Furthermore, they need not acquire any locks. In some cases, readers even do not need to use atomic instructions and memory barriers. Some implementations even need not access any shared data (as far as the locking API itself is concerned).

As Paul McKenney points out, the lack of atomic instructions, memory barriers and shared data accesses is crucial for SMP scalability. [3] According to McKenney's benchmarks, cache misses and pipeline stalls can waste hundreds of times more clock cycles than an average CPU instruction would take. Pipeline stalls are related to memory barriers and atomic instructions. Cache misses are related to accessing (non-constant) shared data, no matter if atomic instructions are used.

1.1.3 RCU Reclaimers

Reclaimers can run either in the same context as writers, or in a separate thread. When a writer replaces a part of a data structure (such as a linked list item) with a new one, there can still be readers accessing the original list item. It is necessary to wait for a certain period of time before the removed data item can be freed, so that all concurrent readers are guaranteed to access valid data. This period of time is called a *grace period.* [11] A grace period is a time interval during which all possible readers pass through a *quiescent state* at least once. A quiescent state of a reader is a moment when no references to the shared data structure are kept. More precisely, all of the reader's local variables possibly containing copies of the shared data are *dead*, meaning that they will not be read before (re-)fetching their values from the shared memory. [3]

The definition of reclaimers summarizes the main purpose of the RCU infrastructure: Observation of quiescent states and detection of grace periods based on the observation. Algorithms based on RCU are useful in read-mostly scenarios. They make the situation slightly more difficult for writers, who have to care about synchronization of their accesses and cooperate with reclaimers. On the other hand, RCU favors readers to „the logical extreme“. [14] (For example, RCU readers need not perform any locking operations whatsoever, not even a simple assignment, when they run on a non-preemptive kernel and are not allowed to block.)

1.1.4 Semantics of RCU

To sum up, **atomic assignment** and **deferred destruction** guarantee that readers always see consistent data and writers can run concurrently with readers. Readers never need to block.

The definition of grace periods implies that extending the grace period interval also yields a grace period. Grace periods detected by different reclaimers on the same system may overlap. Grace period detection does not impose a rendezvous. Quiescent states can occur at any time within the grace period and they may or may not overlap. (Theoretically, quiescent states can have zero duration.)

Last but not least, it is important to note that RCU does not depend on type-safe memory in any way. That said, reclaimers can free removed data as soon as a full grace period elapses, using any standard memory allocator. The corresponding memory area can be reused for a different purpose.

The theoretical background of RCU and numerous design patterns related to both RCU and algorithms based on it have been described in detail by McKenney. [3]

1.2 RCU and Locking

RCU is optimized for situations where readers prevail over writers. Unlike locking, RCU does not enforce mutual exclusion. Readers can run concurrently and writers can run concurrently with readers. It is the writers' responsibility to use other means of synchronization and keep the data structure consistent.

For example, readers-writer locks have some similarity with a special version of RCU. In this „poor man's RCU", all quiescent states are forced to overlap and last until the end of the writer's operation. This has a couple of disadvantages. Unlike real RCU, readers may block. This approach is therefore prone to deadlock scenarios. On the other hand, the grace period detection takes precisely zero time (in theory), since the reclaimer can proceed immediately after the last reader has released its read-side lock.

Despite the numerous advantages, RCU has some important limitations. It is not a drop-in replacement for other means of synchronization, no matter if they are lockless or blocking. It can be used directly in some specific situations, but most algorithms must be modified in order to tolerate some unusual situations that could never occur with mutual exclusion. In some cases, mutual exclusion might be desirable. More importantly, RCU readers can access *stale* data items that are about to be removed or have already been removed from the publicly accessible data structure during the time the reader accesses them. Most real-life algorithms converted to benefit from RCU, such as the directory entry (on Linux) or concurrent hash tables (on K42) had to be modified so that they can safely tolerate stale data. [3, 19].

1.3 Basic RCU Interfaces

This is the standard interface used in most RCU implementations, presented in the form similar to the prototype UTS version of RCU. The RCU API has been implemented in many different forms and first formally described by Paul McKenney. [14] Some operating system kernels use a similar mechanism called *generations*, referring to the fact that multiple versions of each data item can exist (and be accessed by readers) at any given time.

- `rcu_read_lock()` starts a read-side critical section. These calls can be nested. The function never blocks. Its complexity ranges from no-op to atomic instructions and memory barriers, depending on the type of RCU algorithm.
- `rcu_read_unlock()` terminates a read-side critical section. The protected data must not be referenced after this call.
- `rcu_synchronize()` waits until at least one full grace period has elapsed. This call is used when writers and reclaimers execute from the same context. It separates the writer code from the reclaimer code.
- `rcu_call()` enqueues a callback function that will be executed no sooner than after the end of the following grace period. This call is non-blocking and allows writers and reclaimers to run independently in different contexts. Writers may not need to block at all when using the callback API.
- `rcu_call_synchronize()` waits until all currently running and waiting callbacks finish. This call can be used to ensure that no callbacks are accessing a shared data structure that is about to be freed. It is a slow-path call that should be avoided whenever possible.
- `rcu_access()` documents the fact that a variable has to be accessed by the rules of RCU. It may include memory barriers on some CPU architectures [3, 5, 6]. However, it is important to note that OpenSolaris does not run on such architectures.
- `rcu_assign()` documents the fact that a variable has to be accessed by the rules of RCU. It includes a memory barrier to make sure all preceding stores become visible before the variable assignment. (The variable might represent a pointer or a lock.)

Chapter 2

RCU Algorithms Classification

RCU has been implemented in a number of both production and research kernels (IRIX, K42, Linux). [2, 3] Many different forms of the RCU mechanism have been discovered. Some of them track CPUs as read-side entities, whereas others track threads. Both approaches have their specific advantages and drawbacks, described in this chapter. Most algorithms are designed for shared-memory SMP systems. This thesis does not include algorithms for distributed environments.

2.1 CPUs as Readers

This section describes the best known type of RCU, which tracks CPUs as readers. This means that only one reader can run on a CPU at any given time and readers must not block, sleep or be preempted. On the other hand, this form of RCU is very flexible in terms of batching callback requests, which completely separates writers, readers and reclaimers. Writers and reclaimers can run in parallel and in different contexts.

2.1.1 Toy RCU with Forced Preemption

This illustrates the basic concept of RCU. Readers need to disable preemption during their critical sections. Disabling preemption is a simple assignment of a CPU-local variable. Grace periods are detected by forcing a thread to be context-switched on and off all the available processors.

The prototype RCU implementation in the UTS kernel, presented in this thesis, is based around a similar principle. However, unlike the trivial algorithm, the UTS implementation provides a multithreaded callback mechanism with separate reclaimers. Reclaimers can wait for grace periods and handle batches of callbacks in parallel. The algorithm also avoids the need for forced context switches by observing CPU context switch counters, CPU states and the occurrence of RCU API calls. Context switches are used only as a means of forcing a quiescent state when all other detection techniques fail. Unlike the Linux implementation, the UTS RCU is completely independent of clock tick processing and does not require any dispatcher instrumentation.

The new RCU implementations in UTS are described in detail in Chapter 5.

2.1.2 The Classic Linux RCU

Although this implementation is not part of the latest Linux kernels any longer, it clearly demonstrates the ideas behind RCU. It is based on observing naturally occurring quiescent states, such as context switches. Readers are not allowed to sleep or block, which guarantees that context switches imply quiescent states.

Processors requiring RCU attention (those that have unprocessed callbacks in their local queues) raise a flag that can be seen by other CPUs. Quiescent states are detected by observing the number of context switches on the current CPU. When all CPUs reach a quiescent state, a grace period ends and waiting CPU-local callbacks can be processed by each CPU as needed.

A global counter of elapsed grace periods is maintained. Processors compare the counter to their local snapshot and process callbacks based on the number of observed grace periods. All the passive RCU processing, such as quiescent state requests, quiescent state announcements, grace period detection, grace periods counting and callback processing is initiated from the clock tick handler and from the scheduler (instrumented to handle quiescent state announcement on each context switch).

Most data structures are allocated per CPU and each CPU only handles callbacks enqueued by threads that were running on it in the moment of the callback request. This reduces the amount of locking and atomic instructions, since processors always observe their own operations in the code order. [5] Furthermore, callbacks are likely to access locally cached data.

All callback functions are eventually invoked from a tasklet, which is a short-lived CPU-bound execution entity running in a software interrupt context. There have been multiple similar implementation experimenting with other execution entities, but none of them was accepted into the mainline kernel.

The common characteristic of all the RCU implementations derived form this one is the „peer-to-peer“ relationship of processors. Any processor can advance the RCU state machine by either requesting a grace period observation or reacting to other processors' requests. There is no processor and no execution entity with special purpose or privilege.

As with any other synchronization algorithm, the design and implementation of the Classic RCU are complex matters that can not be explained within the scope of this thesis in detail. Readers interested in the the Linux RCU internals are encouraged to read the appropriate chapters of Paul McKenney's dissertation. [3]

The Classic RCU algorithm suffered from a famous race window related to read-side critical sections inside interrupt handlers running on offline or idle processors. The RCU implementation in UTS has a problem similar to this one, but related only to the special case of offline CPUs handling interrupts. Details and proposed solutions are described in section 8.1.

2.1.3 Overlapping Grace Periods and the Counter Ring

The counter ring implementation attempts to eliminate the need for a global „giant“ mutex used by the Classic RCU algorithm. CPUs observe and increment a ring of counter pairs on each quiescent state they pass through.

In the simplest variant of the algorithm, there is only one counter (instead of a counter pair). The current processor's local counter is incremented to the neighbor's counter value plus one by the scheduler and by the the idle loop when there are pending callbacks. A shared atomic variable is used as a counter of pending callbacks.

When the message conveyed by counter increments circulates around all the processors, a grace period boundary is observed by the current processor. Two grace period boundaries (two full ring circulations) guarantee that a full grace period has elapsed. Each processor keeps two lists of callbacks, which is an approach similar to other grace period detection algorithms. One of the lists contains callbacks waiting for the nearest grace period boundary. Callbacks on the second list will have to wait for the subsequent boundary.

The algorithm has been enhanced [3] to support CPU hotplugging. A single counter has been replaced with a pair of counters. The first member of the counter pair is set to the neighbor's first member plus one. The second member is set to the new value of the first one plus the number of active CPUs. A grace period boundary occurs once the neighbor's first counter exceeds the value of the current CPU's second counter. This prevents new processors from causing grace periods to end prematurely.

This algorithm is important from the theoretical point of view: Firstly, as many overlapping grace periods as the number of CPUs can be observed during each ring circulation. This avoids the possible „thunderig herd“ problem that could occur when grace periods do not overlap and callbacks are handled in systemwide bursts. Secondly, the way this algorithm handles dynamic CPU reconfiguration is not only lockless, but also much less complex than other implementations. Most other RCU algorithms need up to hundreds of lines of purely technical code to handle CPU hotplug events, whereas the counter ring algorithm works without even responding to these events explicitly.

With the advent of dynamic clock ticks in CPU idle mode, most advantages of the original algorithm were lost. Current Linux RCU algorithms are designed to avoid any globally shared data (such as a shared atomic callback counter) and to restrain from waking up idle processors. The counter ring algorithm (in its basic form) does not meet these requirements.

The counter ring algorithm was designed and implemented by Rusty Russell. [1]

2.1.4 Hierarchical RCU

This type of RCU can be found in today's Linux kernels. It reduces lock contention by removing the RCU giant lock and introducing a tree-like locking hierarchy with at most four levels. CPUs are represented by leaves of a tree, stored in a compact array. The implementation is tuned so that at most 64 CPUs compete to obtain a lock related to

RCU. When a lock in an internal node is obtained and a grace period (confirmed by all inferior nodes) is detected, the message is propagated up the tree.

This RCU algorithm includes many complicated technical details to deal with CPU offlining, NMI handlers and other kernel mechanisms related to SMP systems. Linux uses dynamically scheduled clock ticks on idle CPUs. Since most (if not all) global RCU algorithms on Linux were originally based on clock tick handlers and the context switching code instrumentation, it was a technically challenging task to guarantee bounded grace periods even in situations where most CPUs are idle. Presumably, interrupt handlers and NMI handlers running on idle CPUs need a possibility to announce their activity, so that read-side critical sections can safely occur in their code.

The way callbacks and grace period announcements are handled for offline CPUs is very similar to the algorithms designed for UTS and presented in this thesis. *Orphaned* callbacks (enqueued on an inactive CPU) can be „adopted“ by any other CPU. (In the UTS implementation, offlined CPUs are always assigned one online CPU to which their callbacks will be relayed. This can form a singly linked list of CPUs when more of them are offlined, but the last member of the list is always an online CPU capable of handling callbacks.)

Interestingly, callback processing includes batch size throttling to avoid long-running batches that would cause noticeable delays. (On UTS, all callbacks are handled from standard thread context rather than interrupt context. Since the callback handling threads are all members of the *System Duty Cycle* scheduling class, there is no need for explicit batch size throttling.)

RCU readers can be made preemptible, which yields a modification of the algorithm that would fall into the „Threads as Readers“ category. This is useful especially in real-time kernels. In fact the hierarchical RCU structure combines multiple RCU algorithms to benefit from their specific features and share data among them.

The hierarchical RCU algorithm does not require idle processors to be woken up on each grace period detection. This removes yet another drawback and scalability limitation of the classic RCU algorithm. (The RCU implementation in UTS never needs to wake up idle processors, since idle processors handling interrupts can be safely distinguished from idle processors doing nothing at all.)

Although hierarchical RCU solves most of the problems related to using RCU from interrupt contexts, it is worth noting that IRQ handlers, NMI handlers and all code dealing with CPU idle states must be instrumented to announce RCU state changes. For IRQs and NMIs, these computations occur no matter if the handler actually contains an RCU read-side critical section or not. Some of these issues and possible solutions are discussed in section 8.1.

2.1.5 Tiny RCU for Uniprocessors

Some Linux developers requested a small and efficient RCU algorithm that would only work on uniprocessors. Although uniprocessors are hard to find among contemporary laptops, desktops or servers, many embedded devices still use this architecture. The

RCU algorithm for uniprocessors has a smaller memory footprint and works faster than other types of RCU on the same hardware. [18]

The „bloatwatch“ RCU algorithm exploits the fact that every quiescent state of the (one and only) CPU is a grace period boundary. A simple linked list of callbacks is maintained and three pointers are used to access it: (1) a pointer to the beginning, (2) a pointer to the end of the current batch and (3) a pointer to the end of the list.

When a grace period ends, which happens each time the scheduler code runs, a tasklet is scheduled to handle the callbacks. It runs all callbacks starting at the first pointer up to the second pointer. Callbacks beyond the second pointer may have been enqueued from the code interrupted by the tasklet, which means they can not be executed immediately and must wait for a subsequent grace period boundary. The second pointer is then set to the value of the third one. The first pointer is set to point at the first unhandled callback.

No atomic operations are necessary to keep the list consistent. Disabling interrupts is sufficient to protect the list when new elements are appended. Integration with the dynamic tick idle mode is much simpler than in hierarchical RCU. This is not surprising, since when the only processor runs idle, there are no other processors that could possibly miss read-side critical sections occurring in interrupt handlers. NMI handlers need not be considered at all in this case.

2.2 Threads as Readers

The introduction of real-time and tickless kernels required new RCU implementations where preemption or even sleeping in read-side critical sections would be possible. Furthermore, it is sometimes useful to detect quiescent states and grace periods locally, without changing (or depending on) the state of threads and CPUs not involved in RCU processing and unrelated to the protected data. All this requires every single RCU reader thread to be tracked separately.

2.2.1 Toy RCU with Readers-Writer Locks

This is a trivial (and severely limited) implementation that illustrates one of the basic ideas of RCU — the grace period detection. However, it does not provide any of the major advantages of scalable RCU implementations. First, readers may block. The basic requirement behind the RCU API definition is to guarantee non-blocking readers. This toy RCU implementation may block, which makes it virtually impossible to use read-side critical sections in interrupt contexts. Second, RCU should allow readers to start and finish independently, so that read-side critical sections running on different processors (or in different threads, when considering preemptible RCU) can overlap with no limitations. This implementation introduces a barrier point that „stops the world“ each time a grace period needs to be detected.

Readers lock and unlock the read-side lock. Writers acquire the write-side lock and release it immediately. All the readers possibly accessing stale data will have finished once the write-side lock is acquired. All quiescent states are forced to occur at the same time. The writer code acts as a barrier before which all the preceding readers will have finished and for which all the subsequent readers will have to wait.

This toy implementation has been introduced into the UTS kernel by the accompanying project of this thesis in order to compare the performance of various RCU implementations to this naïve approach.

2.2.2 Preemptible RCU and the State Machine

Making RCU read-side critical sections preemptible can improve the overall responsiveness of a heavily loaded or real-time kernel. On the other hand, readers are not bound to CPUs any more, so a different approach must be taken when detecting grace periods. [10]

This RCU implementation uses a pair of per CPU counters. Readers increment them when they start and decrement them once they finish. There is no need for atomic instructions and the use of per-CPU data avoids cache misses caused by shared data manipulation. The data structures include a shared global grace period counter and pairs of CPU-local counters. The global counter is manipulated by the grace period detection mechanism. The CPU-local counters are incremented and decremented by the readers as they enter and exit critical sections. The value of the global counter is used to decide which of the CPU-local counters will be used by a reader.

The grace period detection works in four steps. The global grace period counter is incremented and all processors must confirm that they have seen the increment. (1) Based on the global counter increment, all read-side critical sections that start subsequently will manipulate a different set of per-CPU counters than the ones read by the detector. The original counters can now be only decremented. Some of them can be even decremented below zero, due to the fact that readers can migrate randomly. However, their sum will eventually reach zero. (2) Once this happens, all CPUs are asked to issue a memory barrier and announce it. (3) Once all announcements are observed, a grace period has been detected. (4)

The RCU read-side API is extended so that readers use the same counter index when they start and when they finish. The `rcu_read_lock()` function returns the index and the `rcu_read_unlock()` function accepts it as an argument.

The whole four-stage algorithm runs on all CPUs in parallel and is entered from the clock tick handler. Key parts are serialized using a spinlock. Any CPU can advance the state machine if it detects that all preconditions have been met.

Readers are never forced to issue a memory barrier. On CPUs that aggressively reorder memory accesses, this could reorder load instructions before the beginning or past the end of a read-side critical section. [3, 5, 6] This is why the grace period detection algorithm forces all CPUs to issue a memory barrier in the last step.

As described in detail in Paul McKenney's paper, [10] this algorithm could (in theory) allow two global counter flips to occur during one single read-side critical section. This might happen on out-of-order CPUs once they acquire stale data (that could be reclaimed at the end of the current grace period) by reordering a `load` operation to a moment before `rcu_read_lock()` actually fetches the global counter and increments the corresponding local counter. The race condition occurs if the global counter changes between the reordered early `load` operation related to RCU-protected data and the `load` operation fetching the global counter in `rcu_read_lock()`. As a consequence, observing two global counter flips might not guarantee that a whole grace period has elapsed. Moreover, the latest references to RCU-protected data might be reordered to occur behind `rcu_read_unlock()` and the local counter decrement operation.

To cope with these improbable, yet possible situations, the algorithm requires three global counter increments to occur between the time a callback has been added and the moment it can be executed. Paul McKenney gives a proof of correctness based on the periodic memory barriers the grace period detector asks all CPUs to issue. [10]

This algorithm is an interesting illustration of how memory barriers are used (and misused) in various synchronization primitives. For example, an algorithm based on readers-writer locks always contains two barriers per read-side critical section, no matter if there are writers or not. This could degrade performance, since memory barrier instructions are at least one order of magnitude more expensive than standard loads and stores done by RCU read-side primitives. The preemptible RCU algorithm only forces memory barriers to occur when a grace period is actually waited for. Readers do not have to issue these barriers explicitly.

The global RCU implementation for UTS, presented in this thesis, uses a similar notion of „on-demand memory barriers“ to optimize grace period detection. Readers only issue a memory barrier when they observe a grace period counter flip. This approach helps to avoid forced context switches when writers are active, but removes the need for memory barriers completely during periods with no writers.

2.2.3 Sleepable RCU

The original sleepable RCU algorithm introduced into the Linux kernel uses a mechanism very similar to preemptible RCU described above. There are two per-CPU counters incremented and decremented by readers. The counter index is flipped on every grace period request, so that one of the counters can only be decremented from that moment on. This follows from the fact that any subsequent read-side critical sections will increment (and decrement) the other counter, whereas currently running critical sections will decrement the current (observed) counter once they finish. Differences between the preemptible and sleepable RCU algorithms are summarized in the following paragraphs.

First, this algorithm requires readers to issue memory barriers. This adds some overhead on the read side, but the reclaimers can be much simpler. Readers can not access data speculatively before reading the global counter index and incrementing their local counter or after decrementing their local counter. This is why two global counter flips always imply that a full grace period has elapsed.

Second, there is no callback API. This algorithm only implements the blocking API. Sleeping readers could cause an inordinate amount of pending callbacks to accumulate if writers did not have to block. This is why writers and reclaimers must run in the same thread and wait after the writer phase.

Third, the sleepable RCU implementation does not provide a global state machine. Instead, multiple instances of the RCU mechanism with completely separate grace period detection can be created on demand. This makes it possible to keep the grace period detection local, tightly bound to a subsystem, a CPU locality group or a process group. Tracking sleeping readers and detecting grace periods globally could become a scalability issue even with tens of CPUs.

The Linux kernel implements a strong synchronization primitive that waits until all currently running code sections with disabled preemption terminate. The SRCU algorithm is based around this mechanism. Readers disable preemption when manipulating the counters. It is therefore guaranteed that all the subsequent readers will see all preceding changes to global data once all CPUs with disabled preemption re-enable it.

2.2.4 QRCU — Sleepable RCU Friendly to Writers

The QRCU algorithm has been designed by Oleg Nesterov in cooperation with Paul McKenney, [9] who introduced a lockless fast path improvement. The improved QRCU algorithm has also been formally verified by McKenney, using the Promela language.

QRCU is optimized to decrease the overhead of writers (when compared to other RCU implementations) at the cost of slightly increasing the readers' overhead.

The basic principles are very similar to the Sleepable RCU algorithm. There is no `rcu_call()` functionality and no global RCU state. However, there is only one global counter pair instead of local per-CPU counter pairs. This counter pair is incremented and decremented by readers in a way very similar to Sleepable RCU. Presumably, atomic instructions are required to manipulate the shared counters.

Initially, the current counter (referenced by a global index) is set to 1 and the second one is initialized to 0. During the grace period detection, the new counter is incremented, followed by flipping the global index and decrementing the old counter. Once the old counter reaches zero, the current grace period has ended.

Writers can follow two carefully designed fast paths. One of them does not include any locking at all. Both of them do not require the counters to be flipped. Only when these two possibilities fail will the slow path described above be taken. The fast paths can only occur under special circumstances when either no readers are active or a well-defined race with other reclaimers is detected. A description of the algorithm and a proof of correctness can be found in McKenney's paper on QRCU verification. [9]

Starting readers can only increment counters that have a nonzero value. Otherwise they spin, re-reading the global index and retrying the increment operation until they atomically increment a value that was originally nonzero. This avoids the need for memory barriers on the read side. When a reader obtains an old global index speculatively, it may still increment a nonzero value, which is benign, just prolonging the grace period. It is vital that a reader never increment a counter that has already reached zero. Despite the fact that the old counter is not guaranteed to decrease monotonically, it is guaranteed to remain zero once it is decremented to zero.

Finishing readers decrement the same counter they originally incremented, which is enforced using the same read-side API extension as in the preemptible and sleepable RCU variants described above.

The QRCU algorithm has been ported to the OpenSolaris kernel by the author of this thesis to provide an alternative to the global RCU implementation. Chapter 7 describes a benchmark of multiple RCU algorithms running in the UTS kernel.

Chapter 3

RCU Implementation in Linux

This chapter gives a brief summary of the RCU algorithms used in the Linux kernel. The original RCU implementation was enhanced and extended during the past couple of years. Support for sleeping readers has been added and attempts were made to provide bounded grace period duration where expedient callback handling is necessary. Both of these features are vital in realtime kernels.

3.1 History of RCU in Linux

The first Linux RCU implementation appeared in the 2.5 kernel. (There had been RCU patches for 2.4 kernels, but 2.5 was the first mainline kernel to include RCU.) During the years to come, at least five alternative implementations emerged. Once the first implementation, „RCU Classic“, was accepted into the kernel, some key subsystems and algorithms were modified to benefit from RCU. Some of them are listed in section 3.4, but this is definitely not a complete list. RCU has been mostly applied where existence locking was required and where read-mostly data structures needed a scalability improvement.

Requirements specific to real-time kernels lead to the development of preemptible and sleepable variants of RCU. The original preemptible RCU algorithm is not a part of the kernel any more, but many of its original ideas are used in the current tree-based hierarchical RCU code. The sleepable algorithm has remained part of the mainline kernel up to the time of this writing.

3.2 RCU Algorithms Used in Linux

The hierarchical RCU algorithm (see subsection 2.1.4) has been introduced in 2008 by Paul McKenney. [16]. It is currently the default RCU algorithm used in the Linux kernel. It replaced the original „RCU Classic“ algorithm (described in subsection 2.1.2). Linux includes multiple variants of the algorithm, useful for real-time kernels, uniprocessor machines and other special situations. As already mentioned in subsection 2.1.4, this algorithm is supposed to scale to thousands of processors. Contention is reduced by using a hierarchical locking structure.

A special RCU implementation that can handle blocking readers and can be instantiated multiple times to keep grace period detection local has always been kept in the Linux kernel. Currently, the standard sleepable RCU (as described in subsection 2.2.3) is the algorithm of choice. It is used rather rarely. Among the users of sleepable RCU, the BtrFS file system and the KVM code (both host and guest) are the best known ones.

3.3 Other RCU Algorithms

Multiple different flavors of RCU have been implemented in the Linux kernel, based on research in the area and on the needs of some special projects, such as the real-time extensions. [3] Numerous experimental implementations were either not accepted into the mainline kernel, or removed after a few releases.

The counter ring algorithm (described in subsection 2.1.3) and the original preemptible RCU algorithm based on a global state machine (described in subsection 2.2.2) are the best known examples. There have been experimental algorithms testing and benchmarking callback handling from various execution entities and contexts (theads, tasklets, interrupt handlers, scheduler code). [3] Readers interested in the details of the alternative algorithms are encouraged to read Paul McKenney's notes on the RCU API and other papers referenced there. [14]

The QRCU algorithm has been originally discovered by Oleg Nesterov and published on the Linux kernel mailing list. It is a simplified (and very elegant, from the author's point of view) implementations of the original Paul McKenney's sleepable RCU algorithm. (Both algorithms are described in subsections 2.2.3 and 2.2.4, respectively.) Despite being „ack'd“ by Paul McKenney, QRCU has never been accepted into the mainline kernel.

3.4 Use of RCU in Linux

Numerous core kernel algorithms were migrated from mutual exclusion and readers-writer locks to RCU. Most of them had to be transformed so that they can tolerate stale data and other situations specific to RCU, which can not occur under mutual exclusion. These transformations are described by Paul McKenney. [3] McKenney also gives a formal description of design patterns that can be applied to make algorithms „RCU-tolerant“.

RCU is used extensively in code where readers prevail over writers, such as network protocols, network interfaces and packet filtering tables. For example, the route cache or data structures describing VLANs, virtual interfaces and IP address configurations have one common characteristic: Millions of packets might be transferred between two modifications of these data structures. It is therefore beneficiary to allow readers to proceed as fast as possible, without using expensive atomic instructions, at the expense of slightly slower updates. Since these updates are rare, this approach leads to measurable performance benefits. [3] Furthermore, the fault-tolerant nature of most network protocols makes it easy to cope with stale data and other RCU-specific phenomena.

Exploring the current Linux source code (version 2.6.34.1 at the time of this writing) revealed the fact that RCU is becoming increasingly popular. Besides the example algorithms described by Paul McKenney, uses of RCU can be found in numerous subsystems, namely:

- core kernel algorithms (scheduler, CPU related data, performance events)
- virtually all levels of memory management
- auditing and accounting
- security subsystems, SELinux
- the `procfs` pseudo-filesystem
- I/O schedulers, especially CFQ
- the DM (LVM2) subsystem
- the VFS infrastructure (generic file system cache, directory entry cache, event framework)
- some file system drivers (`reiser4`, `ext4`, `btrfs`)
- the KVM virtualization subsystem, both host and guest
- core input device drivers
- some network adapter drivers (`ath9k`, `iwlwifi`)
- virtual network device drivers (TUN/TAP, VLAN)
- NFS and NFSv4 implementations
- most network protocol implementations

All the subsystems mentioned above have a counterpart in UTS. Most of these subsystems could be improved by introducing RCU into the UTS kernel. But as already mentioned, RCU is not a drop-in replacement for readers-writer locks or any other types of locking. Replacing mutual exclusion with RCU always requires careful consideration of all the possible states that could arise from the absence of mutual exclusion and take measures to handle these new states gracefully. A cautious incremental approach with thorough testing is required.

Unfortunately, the unit tests used internally by Oracle are not publicly available. It is therefore practically impossible for an individual outside Oracle to migrate a kernel subsystem from mutual exclusion to RCU and test the new algorithm properly.

Chapter 4

Synchronization and Scheduling in UTS

This chapter lists selected UTS built-in synchronization primitives, as described in Solaris Internals. [7] The listing is only limited to facts relevant to the RCU implementation. Readers interested in a more detailed description can read the chapters on synchronization in Solaris Internals. Similarly, facts about scheduling mentioned in this chapters are limited to kernel processes and one specific scheduling class, the *System Duty Cycle*. More details about scheduling classes and the behavior of kernel processes can be found in Solaris Internals.

4.1 Overview of the UTS Synchronization Primitives

Although read-side critical sections of RCU are completely non-blocking in most implementations, writers mostly use heavy-weight synchronization. The prototype RCU implementation uses the standard high-level locking mechanisms implemented in the UTS kernel to implement its write-side functions. This section lists the most frequently used synchronization primitives and their interfaces.

4.1.1 Mutexes and Condition Variables

Both mutexes and spinlocks are available using the same API in the UTS kernel. By default, all mutexes are adaptive. When explicitly required at initialization or when initialized from an interrupt context, a mutex will become a spinlock.

Adaptive mutexes can spin or block based on the state of the mutex owner. When the owner is on a processor, waiters will spin for some time. Else they will block. When blocking, mutexes use *turnstiles*, a special form of sleep queues that implement priority inheritance. A thread holding a mutex will have its priority increased to the highest priority found on the list of waiters.

When a mutex is unlocked, all the waiting threads are woken up and have to compete again. (Condition variables provide an API that can control this behavior more precisely.)

```
#include <sys/mutex.h>

void mutex_init(kmutex_t *, char *, kmutex_type_t, void *);
void mutex_enter(kmutex_t *);
int mutex_tryenter(kmutex_t *);
void mutex_exit(kmutex_t *);
int mutex_owned(const kmutex_t *);
struct _kthread *mutex_owner(const kmutex_t *);
void mutex_destroy(kmutex_t *);
```

Figure 4.1: The mutex API

```
#include <sys/condvar.h>
#include <sys/mutex.h>

void cv_init(kcondvar_t *, char *, kcv_type_t, void *);
void cv_wait(kcondvar_t *, kmutex_t *);
clock_t cv_timedwait(kcondvar_t *, kmutex_t *, clock_t);
clock_t cv_reltimedwait(kcondvar_t *, kmutex_t *, clock_t, time_res_t);
void cv_signal(kcondvar_t *);
void cv_broadcast(kcondvar_t *);
void cv_destroy(kcondvar_t *);
```

Figure 4.2: The condition variable API

Figure 4.1 shows the mutex API functions. Unlike semaphores, mutexes are based around the notion of the *owner* thread. Names of the API functions are self-explanatory. The last two arguments to `mutex_init()` distinguish between standard mutexes and spinlocks. (In fact only the very last argument is used by the current implementation.) Further details can be found in the UTS source code comments.

Figure 4.2 lists the condition variable API functions. Condition variables in the UTS kernel are very similar to POSIX condition variables. This overview does not list all the API functions. Condition variables provide numerous special variants of the blocking functions that can wait for signals and other events, often related to userspace threads and system call implementation.

4.1.2 Readers-Writer Locks

Readers-writer locks use a special policy to reduce the probability of starvation. This policy can split the set of waiting readers and grant the lock to only some of them when necessary.

Writers have priority over readers when readers are in progress and writers are waiting. In such case, all further readers attempting to enter their critical section have

```
#include <sys/rwlock.h>

void rw_init(krwlock_t *, char *, krw_type_t, void *);
void rw_enter(krwlock_t *, krw_t);
int rw_tryenter(krwlock_t *, krw_t);
void rw_exit(krwlock_t *);
void rw_downgrade(krwlock_t *);
int rw_tryupgrade(krwlock_t *);
int rw_read_held(krwlock_t *);
int rw_write_held(krwlock_t *);
int rw_lock_held(krwlock_t *);
int rw_read_locked(krwlock_t *);
int rw_iswriter(krwlock_t *);
struct kthread *rw_owner(krwlock_t *);
void rw_destroy(krwlock_t *);
```

Figure 4.3: The readers-writer lock API

to block. The lock is then granted to the highest-priority waiting writer once all the remaining readers finish.

When a writer finishes and both readers and writers are waiting, then the readers' priority is inspected. When readers with higher priority than the highest-priority writer are waiting, the read-side lock will be granted to these high-priority readers. The waiting writers (as well as other, low-priority readers) will have to wait for the high-priority readers to finish. Once they finish, a writer will proceed again.

This summary does not list all the possible situations and details on how they are dealt with. Implementation details can be found in both Solaris Internals and UTS source code comments.

Figure 4.3 gives a summary of the readers-writer lock API. Again all the function names are self-explanatory. Lock upgrades from shared to exclusive can only be achieved in a non-blocking manner (using `rw_tryupgrade()`), which avoids the well-known deadlock scenario.

4.1.3 Semaphores

Unlike POSIX semaphores, a kernel semaphore manipulates only one counter at a time. Since semaphores are never owned by a thread (or a group of threads), they do not implement priority inheritance. Threads waiting on a semaphore use a *sleep queue* rather than a turnstile. Semaphores always wake up at most one thread per `sema_v()` operation.

Figure 4.4 lists the semaphore API functions. The `sema_p_sig()` function blocks the thread at an interruptible priority. The return value of 1 indicates that a signal occurred. Else the semaphore has been granted. The `sema_tryp()` and `sema_held()`

```
#include <sys/semaphore.h>

void sema_init(ksema_t *, uint32_t, char *, ksema_type_t, void *);
void sema_v(ksema_t *);
void sema_p(ksema_t *);
int sema_p_sig(ksema_t *);
int sema_tryp(ksema_t *);
int sema_held(ksema_t *);
void sema_destroy(ksema_t *);
```

Figure 4.4: The semaphore API

functions can be used to inspect the state of the semaphore with or without actually trying to decrement it, respectively. Both of them are non-blocking.

4.1.4 Shuttles

A shuttle blocks one thread and borrows its scheduling context to another, waiting thread. This approach is much more efficient than standard sleep/wakeup mechanisms, such as turnstiles and sleep queues.

Shuttles are used to implement *door call* servers. After a server thread is chosen from a thread pool, the shuttle freezes the caller and lets the server thread run on the caller's behalf. The server will be halted again (in a well-defined state) after processing the request and the client will resume.

A shuttle is somewhat similar to a pair of binary semaphores in a producer-consumer scenario, but it is much more efficient than semaphores.

Shuttles are not used by the current RCU implementation, so they will not be described here in detail.

4.2 Other Means of Synchronization

This section gives a basic overview of atomic instructions and memory barriers, both of which are not synchronization primitives in the most common sense. They neither block nor enforce mutual exclusion. However, they are vital for algorithms where blocking has to be avoided, such as RCU.

4.2.1 Atomic Instructions

Unlike Linux, the UTS kernel does not define any „atomic data type“ (which is in most cases just a structure containing a volatile integer). The UTS kernel defines atomic operations as functions that accept a pointer to a standard integer variable as their first argument. Atomic operations are implemented separately for integers of all sizes and for pointers.

```
#include <sys/atomic.h>

void atomic_inc_32(volatile uint32_t *);
void atomic_dec_32(volatile uint32_t *);
void atomic_add_32(volatile uint32_t *, int32_t);
void atomic_or_32(volatile uint32_t *, uint32_t);
void atomic_and_32(volatile uint32_t *, uint32_t);
uint32_t atomic_inc_32_nv(volatile uint32_t *);
uint32_t atomic_dec_32_nv(volatile uint32_t *);
uint32_t atomic_add_32_nv(volatile uint32_t *, int32_t);
uint32_t atomic_or_32_nv(volatile uint32_t *, uint32_t);
uint32_t atomic_and_32_nv(volatile uint32_t *, uint32_t);
uint32_t atomic_cas_32(volatile uint32_t *, uint32_t, uint32_t);
uint32_t atomic_swap_32(volatile uint32_t *, uint32_t);
```

Figure 4.5: Atomic instructions

The RCU implementation uses atomic instructions to manipulate pointers and 32-bit integers. The API listing in figure 4.5 only lists functions manipulating 32-bit integers. All the other functions can be derived from this listing or found in the corresponding header file.

The functions with the `_nv` suffix return the new value. All atomic operation are coded in assembler and can be found in the platform-dependent `atomic.s` files.

4.2.2 Notes on Memory Barriers

The approach to memory barriers is not the same in OpenSolaris and Linux. OpenSolaris has stronger assumptions about memory ordering than Linux. Consequently, memory barriers used by the UTS kernel (and globally defined in OpenSolaris) are much weaker than those used in Linux.

The different approach of the two kernels to memory ordering can be explained by the fact that the UTS kernel relies on the TSO (Total Store Ordering) processor mode, whereas Linux assumes that SPARC CPUs will run in the RMO (Relaxed Memory Ordering) mode, no matter if they actually implement this mode. (As far as the author knows, the most recent SPARC chips only implement the TSO mode.) This is why the memory barriers defined in Linux have to be stronger than those defined in UTS.

Recent Intel processors can run in *out-of-order store* mode. [23] This mode could theoretically allow the CPU to optimize the instruction flow by reordering memory accesses more aggressively.

The current UTS kernel uses Total Store Ordering (or its Intel equivalent) all the time. However, as some OS/Net developers already noted, „this will certainly change in the future“. [13] This means that future-proof algorithms should not rely on the TSO.

```
#include <sys/atomic.h>

void membar_producer();
void membar_consumer();
void membar_enter();
void membar_exit();
void membar_sync();

void membar_full();
```

Figure 4.6: Memory barriers

Some parts of the UTS kernel use a strong memory barrier that forces all operations (even those not related to memory access) to finish before those that occur after the barrier. This barrier is used in code related to hardware drivers and clock tick processing. However, a standard „full“ memory barrier (which forces ordering of all memory accesses, both loads and stores, but does not force all other operations to stop) was not defined so far. The RCU algorithm includes fast-path cases that require such a full memory barrier, through which neither writes nor reads can be reordered. For this reason, a new memory barrier function called `membar_full()` has been introduced. Figure 4.6 lists memory barrier functions.

It is worth noting that most memory barrier considerations are related to SPARC processors. (Similarly, Linux memory barriers are based on Alpha processors, which implement the most relaxed memory model of all the supported platforms. [5] [6]) Unlike SPARC processors that implement 15 fine-grained memory barriers (and some other barriers related to non-memory operations), [17] Intel processors only provide three memory barriers. [6]

Memory barriers around standard mutual-exclusion critical sections in the UTS kernel are based on two important assumptions:

1. When a `load` instruction occurs before a critical section entry point, it is legal to reorder the `load` operation to happen inside the critical section.

2. When a `load` instruction occurs after a critical section exit point, it is legal to reorder the `load` operation to happen inside the critical section.

For mutual exclusion or readers-writer locks, both of these implications hold. The corresponding memory barriers (`membar_enter()` and `membar_exit()`) are defined accordingly, so that they allow the processor to optimize memory accesses. Unfortunately, both optimizations are illegal when it comes to RCU read-side critical sections and some other members of the RCU API. This is the main reason why `membar_full()` had to be introduced.

4.3 UTS Scheduling Features Used by RCU

This chapter describes how RCU benefits from a recently introduced *System Duty Cycle* scheduling class and task queues based on this class.

4.3.1 System Duty Cycle Scheduling Class

The classic Solaris 10 kernel (as described in Solaris Internals [7]) did not offer much flexibility in scheduling kernel threads and processes. Kernel threads were never limited by time quanta. A kernel thread would run until it got preempted by a higher-priority thread, blocked or preempted itself voluntarily. This simple approach was suitable in most cases, as long as the kernel was not required to perform computationally-intensive or batched work.

Computationally intensive kernel threads might occur in ZFS. The filesystem provides transparent checksumming, compression and software RAID features, all of which might require computations on large data. These computations could theoretically delay userspace threads for a very long time, causing delays noticeable by humans. This is why the System Duty Cycle scheduling class had to be introduced. Further details can be found in the UTS source code.

The System Duty Cycle class switches a kernel thread's priority between the lowest possible and the highest possible priority based on the amount of time the thread spends on the CPU. Furthermore, clock tick handler methods of this class can preempt kernel threads of the same class, moving them to the tail of the dispatch queue of their processor and priority. This provides simple implicit time quanta and a round-robin scheduling behavior.

Each thread specifies a percentage of time it needs to spend on a CPU (when not sleeping). If a runnable thread spends more time on a CPU, its priority is dropped to the lowest possible priority, letting other threads (including userspace processes) to run. When the time recently spent on a CPU is too low, the priority of the thread is boosted above all userspace and most kernel threads. This is done by the clock tick handler callback implemented in the system duty cycle class. Presumably, the sum of system duty cycle percentage of runnable threads might (and often does) exceed 100%. The source code comments give a detailed summary of how the scheduling algorithm behaves in such cases.

The project related to this thesis modified the system duty cycle thread initialization so that the „low“ and „high“ priorities can be specified explicitly instead of using the default values of 0 (the lowest priority) and 99 (the highest priority of a standard kernel thread). For example, an algorithm might need a system duty cycle thread whose priority must never exceed the priority of another thread. In fact this is what the RCU implementation needs in order to work properly. Reclaimer threads must never exceed the priority of the detector thread. (These threads are described in chapter 5.)

The RCU implementation uses the System Duty Cycle class for batched callback handling. All threads that handle callbacks are therefore guaranteed to run with a high priority as long as they stay below their duty cycle percentage or there is nothing else to do. On the other hand, they will never block the whole system and cause noticeable delays when processing large batches.

4.3.2 System Duty Cycle Task Queues

Task queues provide a unified implementation of thread pools that can be used to handle various deferred tasks. They are a means of avoiding deadlocks and performing possibly blocking operations from contexts that must not block. Task queues can run under the System Duty Cycle scheduling class. These special task queues are used by RCU to implement the support for blocking callback functions.

Task queues used by RCU are *dynamic*, which means that they create worker threads on demand and avoid contention during dispatching operations by maintaining per CPU data structures and thread pools. (The UTS kernel also contains an older and simpler „static“ task queue implementaion. Task queue type can be chosen at queue instantiation time.)

Unfortunately, the current task queue implementation has two major drawbacks. First, threads entering the SDC class must have an associated LWP. (This in fact applies to threads of all scheduling classes except for the standard „system“ class.) However, the thread pool is populated by raw threads for the sake of efficiency and fast thread spawning. This is why only the main dispatcher thread of the SDC task queue actually runs under the SDC class, whereas the worker threads are standard system class threads. This limitation is likely to be removed in the future, as soon as *microstate accounting* information is moved from LWPs into kernel threads.

The second limitation is related to the blocking nature of the task queue dispatch mechanism. The API does not require the callers to provide an allocated data structure. Consequently, the task queue may need to allocate memory when big bursts of tasks are dispatched. The dynamic task queue implementation uses per CPU buckets to store the task data and manage worker threads, which avoids contention as long as all callbacks can be handled in time. However, once too many callbacks accumulate and most buckets are overfilled, most threads start to compete for a global task queue lock in order to extend their per CPU data structures or relay some work to another bucket. This degrades the performance to a scenario similar to a livelock. Most threads spin on mutexes all the time and useful work is only done by chance. The author observed the rate of executed tasks dropping from tens of thousands per second to a frequency suspiciously close to clock ticks. This might mean that all useful work was only due to the randomness induced by interrupts.

For the reasons mentioned above, it is recommended to use the standard RCU mechanism for non-blocking callbacks wherever possible. This mechanism guarantees full callback handling performance, no matter how many callbacks are added. It also guarantees that all callbacks are always handled by system duty cycle threads.

Chapter 5

RCU Implementation in UTS

The RCU implementation discussed in this chapter has been contributed by the project this thesis is based on. All the implemented variants of RCU and their interfaces are listed. The description focuses on the algorithms rather than technical implementation details. Readers interested in these details are encouraged to read the full source code and comments.

5.1 The Global RCU Implementation

This section describes the system-wide RCU implementation that provides the full RCU API, both blocking and callbacks. It also includes some extra features, such as waiting for all pending callbacks to finish.

5.1.1 Features and Characteristics

A *detector* thread checks the state of processors and provides one global source of grace period events. This means that grace period boundaries are announced globally and never overlap. It is important to distinguish between *grace period boundaries* and *grace periods*. Unlike grace period boundaries, grace periods might overlap, as observed by threads waiting for them to elapse. To detect a full grace period, at least two grace period boundaries must be observed.

Each CPU has a bound reclaimer thread for callback processing. Callbacks can be either processed directly by the reclaimer or offloaded to a task queue. Callbacks processed by the task queue can sleep and block, whereas those processed by reclaimer threads are not allowed to do so. Threads waiting for a grace period to elapse use exactly the same mechanism as the reclaimer threads to communicate with the detector thread.

Reclaimers always wait for one grace period boundary to occur and keep two lists of waiting callbacks — those waiting for the next boundary and those waiting for the subsequent one. This makes it possible to process callbacks as often as possible, on every single boundary observed by the reclaimer. Other threads (calling the RCU blocking API, but unrelated to the RCU internals) always wait until two boundaries are observed, which is necessary to guarantee that a full grace period has elapsed.

Besides the reclaimer threads processing non-blocking callbacks, the RCU implementation includes a possibility to block and sleep in callback handlers. Callbacks that can block are handled separately, using a system-wide task queue. However, as already described in subsection 4.3.2, serious performance issues can arise if callbacks marked as blocking are produced irresponsibly. Producing an inordinate amount (such as bursts of millions) of blocking callbacks is strongly discouraged. Non-blocking handlers should be used whenever possible.

The RCU implementation responds to CPU hot(un)plug operations by initializing or stopping corresponding reclaimer threads and by reallocating the callback handling task queue so that its capacity grows and shrinks with the number of available CPUs. This has some important implications when considering interrupt handlers and interrupt threads running on offline processors. (Such a situation is perfectly legal in UTS.) These details are discussed in chapter 8.

Quiescent states are both observed and forced. Observed quiescent states include the idle thread, user-mode execution, context switches, the CPU offline state and the use of RCU primitives. Some RCU API functions observe the RCU global counter state. When a new grace period is detected (the global counter has changed), they issue a memory barrier and announce the counter state they observed in the per CPU data structures of the processor they run on. This means that no barriers are needed in the absence of writers and in cases when read-side sections start and finish during the same grace period. This opportunistic grace period observation reduces the number of forced context switches on SMP systems. Only processors that did not observe the latest global counter state during the whole grace period (and are neither idle nor in userspace) will need forced rescheduling.

5.1.2 The RCU API

Figure 5.1 shows the interface of the global RCU implementation.

The first two functions start and end a read-side critical section. The `rcu_call*()` family of functions enqueues a callback. Either standard or *exclusive* (sleepable) callbacks can be enqueued. Producing callbacks at high interrupt levels is handled by the two functions containing `excl` in their names. They behave exactly the same way as their „non-exclusive“ counterparts when running on an online CPU. On offline CPUs, they relay the callback to another (online) CPU. (Inserting a callback into an offline processor’s queue would cause the callback to wait indefinitely, as discussed in the following subsections.)

`rcu_synchronize()` and `rcu_synchronize_shared()` are part of the blocking RCU API. They block until at least one grace period elapses.

The `rcu_call_synchronize()` function waits until all callbacks enqueued up to this moment terminate. It is necessary to stress that only callbacks enqueued before this function is invoked will be waited for. There are no guarantees related to callbacks produced by other threads during the time the current thread blocks and waits.

The last function returns the current state of the grace period counter, for the sake of testing and statistics.

```
void rcu_read_lock();
void rcu_read_unlock();

void rcu_call(rcu_cb_t callb, rcu_t *arg, rcu_weight_t weight);
void rcu_call_excl(rcu_cb_t callb, rcu_t *arg, rcu_weight_t weight);
void rcu_call_high(rcu_cb_t callb, rcu_t *arg, rcu_weight_t weight);
void rcu_call_excl_high(rcu_cb_t callb, rcu_t *arg, rcu_weight_t weight);

void rcu_synchronize();
void rcu_synchronize_shared(uint32_t);

void rcu_call_synchronize();

uint32_t rcu_get_gp();
```

Figure 5.1: The RCU API

5.1.3 Read-Side Algorithms

As already mentioned, the global RCU algorithm is based on tracking processors as readers. This means that all read-side critical sections are not allowed to block and have to run with disabled preemption. The weaker (and more flexible) `thread_nomigrate()` and `thread_allowmigrate()` primitives are not sufficient to guarantee correct operation of the RCU algorithms and reasonably bounded grace periods. However, interrupts can stay enabled throughout the whole read-side critical section.

At the beginning of the read-side critical section, preemption has to be disabled first. This is a simple assignment to a CPU-local variable. Once preemption is disabled, the local per CPU grace period counter snapshot is compared to the global one. If the global counter is greater than the local one, the local one is assigned and a full memory barrier is issued. This opportunistic grace period observation brings two important advantages. First, the memory barrier instruction need not be used in the absence of grace period observation activity. Second, observing the grace period provides a means of avoiding forced context switches. The detector thread can find out whether a CPU has already observed the current grace period or not. The fact that the current global counter state has been observed implies that the observing CPU also issued a full memory barrier during the current grace period. This in turns implies that it can not be accessing stale data from the preceding grace period. The last implication tells us that the current grace period can end without forcing the observing CPU to perform a context switch.

5.1.4 Read-Side Code Walkthrough

Listings 5.1 and 5.2 show the source code of the read-side primitives. The `rcu_nesting` variable is a per CPU counter that makes nested read-side critical sections possible. Without the nesting counter, an opportunistic grace period observation could occur too early, at the end of the innermost read-side critical section. Presumably, this could lead to data corruption in many cases. It is important to stress that read-side section nesting can occur both intentionally (in a function call hierarchy, for example) and **unintentionally** (due to read-side critical sections in interrupt handlers). Since the `rcu_nesting` variable is only accessed by the local CPU, it can never get clobbered, even when the standard RISC load—modify—store sequence is used.

Listing 5.1: Beginning of a read-side critical section

```
void
rcu_read_lock()
{
        cpu_t   *cp;

        kpreempt_disable();
        cp = CPU;
        rcu_announce_qs(cp);
        ++RCU_CPU(cp, rcu_nesting);
}
```

Listing 5.2: End of a read-side critical section

```
void
rcu_read_unlock()
{
        cpu_t   *cp = CPU;

        --RCU_CPU(cp, rcu_nesting);
        rcu_announce_qs(cp);
        kpreempt_enable();
}
```

Listing 5.3: The opportunistic grace period observation

```
static inline void
rcu_announce_qs(cpu_t *cp)
{
        ASSERT(curthread->t_preempt);

        if (!RCU_CPU(cp, rcu_nesting)) {
                if (RCU_GP_GT(rcu_cb.rcu_gp_ctr, RCU_CPU(cp, rcu_gp_ctr))) {
                        gp_t    gp_ctr = rcu_cb.rcu_gp_ctr;
                        membar_full();
                        RCU_CPU(cp, rcu_gp_ctr) = gp_ctr;
                }
        }
}
```

Listing 5.3 shows the opportunistic grace period announcement. It is important to read the global counter value before the memory barrier and assign it after the barrier. We must make sure no RCU protected data from the preceding grace period is accessed before announcing the new one (as observed by other processors). Furthermore, we must guarantee that we do not read any data belonging to the new grace period sooner than the global counter is loaded from memory. Else another thread could increment the global counter between the time we obtain the first item of RCU protected data and the time we load the global counter. This would cause the reader to announce a grace period it has not yet observed, as perceived by other CPUS.

All the notes on possible instruction reordering include the (implicit or explicit) „as seen by other CPUs“ epilogue. This topic deserves a couple of notes. Processors always perceive their own operations in the instruction code order. [5] Reordering of memory accesses is a „team sport“ in the sense that multiple parties must be involved for this phenomenon to be actually noticed. This can be either multiple processors or a processor and a memory-mapped device. Furthermore, it is important to stress that a memory barrier is in no way related to global data visibility. Specifically, a store barrier is not a „commit“ operation and a load barrier will not force unfinished memory operations to become visible. A barrier merely instructs a processor to make sure that global visibility of operations that occur before the barrier does not precede the global visibility of those that occur thereafter. It does not enforce any limitations on when exactly the data will be (physically) fetched or stored. For example, if instructions following a barrier include one hour of register-only operations, the processor would be within its rights to postpone some of the memory operations preceding the barrier almost indefinitely, from a machine's perspective.

5.1.5 Write-Side Algorithms

First of all, writers have to make sure their changes will not be reordered unexpectedly. This can be done using the standard `rcu_assign()` macro, by issuing a memory barrier explicitly or by using a non-blocking data structure that protects its data by atomic instructions. (The third approach is taken in the RCU benchmarking code contributed as part of this thesis. Some of the RCU testing code also uses the first method.)

Once ordering issues are resolved, writers have to wait until all the readers that may have observed the old state of the modified data structure finish. Since there is no way to find out which of the currently running readers are using stale data, the easiest way is to wait for all currently existing read-side critical sections to terminate. Again, detecting such an event would be rather complicated. (However, the QRCU algorithm described in section 5.2 in fact does this.) To simplify the detection, writers ask a central grace period detector thread to wake them up as soon as at least one full grace period elapses. This isolates them from the complexity of the actual grace period detection at the expense of a slightly longer grace period duration.

Writers that have removed a data item have two options: They can either wait for a grace period to elapse and become reclaimers, or register a callback handler to do the reclamation work. The second method is non-blocking. The callback handler is guaranteed to be invoked no sooner than after a full grace period.

Callbacks are handled by CPU-bound reclaimer threads. They act just like any other threads waiting for grace period boundaries. As already mentioned in subsection 5.1.1, they wait for every single grace period boundary and keep multiple lists of waiting callbacks. This guarantees that callbacks can be processed on every observed grace period boundary when necessary, but every single callback will wait for at least two boundaries (a complete grace period) to pass by. These threads will be described in detail later on.

5.1.6 Write-Side Code Walkthrough

Listing 5.4 shows the blocking API function. It is used by writers that want to become reclaimers and safely free the resources they had made inaccessible during the writer phase.

Line 6 guarantees that the grace period counter observation will not take place too early, possibly before the last access to the protected data. Lines 10–13 take a short path if this thread has waited long enough to acquire the mutex. Waiting for at least two grace period boundaries guarantees that a full grace period has elapsed and there is no need to interact with the detector thread directly.

The following code on lines 14 and 15 announces a *high water* condition. The exact meaning of this flag is discussed in subsection 5.1.7, which describes the reclaimer code. This causes the detector thread to behave aggressively and announce a grace period boundary as soon as possible. The `rcu_synchronize_common()` function invoked on line 16 waits for two grace period boundaries to elapse.

Listing 5.4: Waiting for a grace period to elapse

```
 1 void
 2 rcu_synchronize()
 3 {
 4         gp_t    gp_ctr;
 5
 6         membar_full();
 7         gp_ctr = rcu_cb.rcu_gp_ctr;
 8
 9         mutex_enter(&rcu_cb.rcu_gp_mutex);
10         if (rcu_cb.rcu_gp_ctr - gp_ctr >= 2) {
11                 mutex_exit(&rcu_cb.rcu_gp_mutex);
12                 return;
13         }
14         ++rcu_cb.rcu_hiwater;
15         cv_signal(&rcu_cb.rcu_hiwater_cond);
16         rcu_synchronize_common(2);
17         --rcu_cb.rcu_hiwater;
18         mutex_exit(&rcu_cb.rcu_gp_mutex);
19 }
```

Listing 5.5 shows the „patient“ version of the RCU blocking API. The basic idea behind the standard `rcu_synchronize()` function is as follows: *There is a thread that can free some resources and continue running, as soon as we detect a grace period. So let us detect it as quickly as possible.* This simplified approach may not be suitable in all cases. Perhaps the thread has a very low priority and blocking it for a long time is not a problem. Or perhaps the thread is bound to exit right after its reclaimer phase, which also does not justify aggressive grace period detection.

The function shown in listing 5.5 takes an argument specifying how many threads can be waiting for a grace period. If fewer threads are waiting, then the grace period detection will be delayed by a constant amount of time, which allows the opportunistic grace period observation to serve its purpose. Once too many threads are waiting, a *high water* state is announced, just like in listing 5.4.

Presumably, this algorithm can be inaccurate in a case when many threads with a high `maxw` value start waiting for a grace period. In such case, nobody will notify the sleeping threads that the number of waiters has been exceeded. Maintaining a maximum value of `maxw` could be a solution, but the problem of updating this value as threads wake up would have to be solved correctly. Resolving this issue is left to future work.

Listing 5.5: „Patient“ waiting for a grace period to elapse

```
void
rcu_synchronize_shared(uint32_t maxw)
{
        gp_t    gp_ctr;

        membar_full();
        gp_ctr = rcu_cb.rcu_gp_ctr;

        mutex_enter(&rcu_cb.rcu_gp_mutex);
        if (rcu_cb.rcu_gp_ctr - gp_ctr >= 2) {
                mutex_exit(&rcu_cb.rcu_gp_mutex);
                return;
        }
        ++rcu_cb.rcu_wait_ctr;
        if (rcu_cb.rcu_wait_ctr > maxw) {
                ++rcu_cb.rcu_hiwater;
                cv_signal(&rcu_cb.rcu_hiwater_cond);
                rcu_synchronize_common(2);
                --rcu_cb.rcu_hiwater;
        } else {
                rcu_synchronize_common(2);
        }
        --rcu_cb.rcu_wait_ctr;
        mutex_exit(&rcu_cb.rcu_gp_mutex);
}
```

As already mentioned, the `rcu_synchronize_common()` function communicates with the detector thread. It is shown in listing 5.6. Lines 9–12 wake up the detector thread when necessary. Lines 14–16 wait until the requested number of grace period boundaries elapses. Since threads can be waiting for either one or two boundaries to elapse, two condition variables are used to announce them, in order to avoid spurious wake-ups.

Listing 5.6: Blocking for a number of grace period boundaries

```
1 static void
2 rcu_synchronize_common(gp_t periods)
3 {
4         gp_t    gp_ctr = rcu_cb.rcu_gp_ctr + periods;
5
6         ASSERT(MUTEX_HELD(&rcu_cb.rcu_gp_mutex));
```

```
        ASSERT(RCU_CPU(CPU, rcu_nesting) == 0);

        if (rcu_cb.rcu_gp_wanted < periods) {
                rcu_cb.rcu_gp_wanted = periods;
                cv_signal(&rcu_cb.rcu_wanted_cond);
        }

        while (RCU_GP_GT(gp_ctr, rcu_cb.rcu_gp_ctr))
                cv_wait(&rcu_cb.rcu_elapsed_cond[gp_ctr & 0x1],
                    &rcu_cb.rcu_gp_mutex);
}
```

The basic form of the RCU callback API is shown in listing 5.7. Line 4 initializes the RCU structure, which must be allocated on the heap and provided by the caller. Exclusive callbacks are added the same way, but initialized with the `RCU_CB_EXCL` value. The other two forms of the callback API (which can be called from a high interrupt level) are similar. They include some technical code to check whether the current processor is offline and take the necessary steps if so.

Listing 5.7: Producing RCU callbacks — the interface

```
 1 void
 2 rcu_call(rcu_cb_t callb, rcu_t *arg, rcu_weight_t weight)
 3 {
 4         rcu_init_arg(callb, arg, weight, RCU_CB_DEFAULT);
 5         rcu_call_common(arg, weight);
 6 }
```

The `rcu_call_common()` function is shown in listing 5.8. Line 6 guarantees that the caller's operations on the shared data will be visible earlier than the first subsequent store operation. Line 7 disables preemption. If we enqueued the first callback to a reclaimer's list on line 9, we must wake up the reclaimer by incrementing a semaphore. (Else someone has already done so.) We then add the weight value to the per CPU weight sum (line 11) and announce a grace period observation when applicable. The meaning of this value will be discussed in subsection 5.1.7.

Listing 5.8: Producing RCU callbacks — the internals

```
 1 static void
 2 rcu_call_common(rcu_t *arg, rcu_weight_t weight)
 3 {
 4         cpu_t   *cp;
 5
 6         membar_exit();
 7         kpreempt_disable();
 8         cp = CPU;
 9         if (rcu_enqueue_callback(cp, arg, &arg->rcu_next))
10                 sema_v(&RCU_CPU(cp, rcu_queue_sema));
11         RCU_ADD(RCU_CPU(cp, rcu_weight), weight);
12         rcu_announce_qs(cp);
13         kpreempt_enable();
14 }
```

Callbacks are enqueued using an atomic operation that advances the list end pointer at the same time. This can be seen in listing 5.9. There is an obvious race window between appending to the list atomically (lines 6–7) and bridging the gap between the new element and the previous end of the list (line 8). This is never a problem when this code runs on the same processor as the reclaimer thread, thanks to the disabled preemption.

The special case when this code is invoked from an interrupt handler running on an offline CPU could theoretically lead to problems. Since the list will not be traversed before at least two grace period boundaries are detected, it is highly improbable that an inconsistent state of the list could ever be observed. Unfortunately, it is still theoretically possible. Resolving this issue is left to future work. Details and suggested solutions are mentioned in section 8.1. The problem is related to the fact that offline CPUs currently can not influence the grace period detection mechanism. This feature is vital for both readers and writers to operate safely from high interrupt level contexts.

Let us emphasize that producing callbacks on online processors is always safe (from any context), since online processors will never observe a grace period before all the list operations are complete, no matter how many interrupts occur and how deeply they are nested. The atomic append operations guarantees that the list is consistent and that callback handling will not take place before the new tail is connected properly. The problem mentioned above is only related to the very special case when a processor is offlined, but its callback handling duties persist for some hardware-related technical reasons.

Listing 5.9: Producing RCU callbacks — appending to the list

```
 1 static inline int
 2 rcu_enqueue_callback(cpu_t *cp, rcu_t *append, rcu_t **append_tail)
 3 {
 4         ASSERT(curthread->t_preempt);
 5
 6         rcu_t   **oldtail = atomic_swap_ptr(
 7             &RCU_CPU(cp, rcu_nexttail), append_tail);
 8         *oldtail = append;
 9         return (oldtail == &RCU_CPU(cp, rcu_nextlist));
10 }
```

5.1.7 Reclaimer Algorithms

Instead of waiting for one grace period and becoming reclaimers, writers can proceed immediately and without blocking. This functionality is implemented using a callback mechanism. The callback handler containing the reclaimer code is guaranteed to be run no sooner than before at least one grace period elapses.

Most callback handlers should be related to freeing memory and these standard handlers never need to sleep. However, it is still possible to think of a handler acquiring a mutex or performing another blocking action. To make this possible, callbacks are handled using two different mechanisms.

Callback handling starts with a per CPU callback list. Callbacks produced on the same CPU are added to the list. Each CPU has a CPU-bound reclaimer thread that takes care of callback processing. When no activity related to RCU takes place, the reclaimer thread sleeps. It is woken up when callbacks are added to its first callback list, called the *nextlist*. The reclaimer thread than asks the detector thread to announce a grace period boundary and waits exactly the same way as other threads waiting for grace periods. When a boundary is announced, all the callback records from *nextlist* are moved to another list, called *curlist* in the source code. And all the former members of curlist are processed. This is how the reclaimer threads make sure each callback waits for at least two grace period boundaries. At the same time, callback processing can take place on every single grace period boundary (when waiting callbacks are available), saving considerable time and effort spent in the grace period detector thread.

Standard callbacks are not allowed to sleep and the reclaimer threads execute them directly. Other callbacks (called *exclusive* in the source code) could potentially sleep and must be processed separately. Instead of executing them, the reclaimer thread inserts them into a system-wide dynamic task queue. This is how callback handlers allowed to sleep can be used without stopping the whole reclaimer thread. Since direct callback handling is several (decadic) orders of magnitude faster than the task queue, it is recommended to avoid sleepable callbacks when possible. On the other hand, the possibility of blocking (and locking adaptive mutexes, for example) in callback handlers might be useful or inevitable in some cases.

Each callback has an unsigned integer value called *weight*. Instead of tracking the number of pending callbacks, the reclaimers watch the weight sum. Each callback can have a different weight (importance), based (for example) on how much memory or other resources it will free. The weight estimate should be computed by the `RCU_WEIGHT()` macro, which takes two arguments — the number of `kmem_free()` invocations and the number of bytes to free. The result is a combination of these two values, so that callbacks that either free a huge number of buffers or free a large area of memory are considered more important.

When the callback weight sum on a processor exceeds a predefined value (which is a tunable value, roughly corresponding to thousands of `kmem_free()` invocations or units of megabytes of memory to be released), the reclaimer thread sets a *high water* mark and signals the detector thread. The high water mark causes the detector thread to behave aggressively, not attempting to wait a couple of clock ticks for the opportunistic counter observation announcements or for naturally occurring context switches. Instead, all CPUs are forced to context-switch to the detector thread. Presumably, some processors will announce a grace period counter observation or pass through a natural context switch during the detector's operation. These processors will not be interrupted in any way during the current boundary detection.

The high water flag speeds up the grace period detector when one or more threads require that a grace period boundary be announced as soon as possible. Reclaimers that currently do not wait for a grace period (which means they are either idle or processing callbacks) have their high water flags cleared. (It would not make sense to announce grace period boundaries expediently when nobody is listening.)

In UTS, writers producing callbacks can run on offline CPUs. An interrupt handler or an interrupt thread running on an offline (yet not quiesced) processor can enqueue an RCU callback. Since the reclaimer threads are standard threads, they never run on offline CPUs. This means that callback processing can not take place on offline CPUs. These issues are resolved by relaying the callbacks to another (online) CPU, chosen when the offline CPU changed its state. A CPU from the local processor partition is looked up. If there is no such online CPU, then the locality group is traversed to find one. If this fails as well, an online CPU is chosen at random. When the CPU to which callbacks can be relayed becomes offline, exactly the same mechanism is used. This means that offline processors are on singly-linked lists terminated by an online processor that receives and handles callbacks produced on its predecessors.

The handover process had to be be carefully designed to avoid losing callbacks produced during the CPU state change phase. This required adding a new CPU state event (`CPU_POSTOFF`) that triggers after a processor has been successfully offlined. By that time, the reclaimer thread bound to the offline CPU does not run any more, which means that all the callbacks that possibly exist on the CPU's *nextlist* become *orphaned*, as expressed in the Linux parlance. (Callbacks from the *curlist* are always handled by the reclaimer thread before it dies.) The CPU to which callbacks will be relayed has been chosen and assigned before actually offlining the offline CPU. (Another CPU event handler does this.) We can therefore be sure that no further callbacks will be enqueued on the offline CPU. (They will be immediately relayed to the chosen online CPU instead.) This implies that the `CPU_POSTOFF` event handler can safely manipulate the offline CPU's *nextlist* and append it to the related online CPU's *nextlist*. This is how the orphaned callbacks are handled.

5.1.8 Reclaimer Code Walkthrough

The reclaimer threads all run the loop shown in listing 5.10. Lines of purely technical code have been omitted from the listing for the sake of readability.

Line 4 checks whether there are callbacks to process. If not, line 6 handles the case when a `rcu_call_synchronize()` request has to be answered. It calls a handler function that decrements a global counter. If this reclaimer was the last one to announce an idle state, it wakes up the thread waiting for this announcement. This procedure is described in subsection 5.1.9.

Line 8 checks whether this reclaimer should die. It may be requested to terminate when a processor is offlined. Lines 10 and 14 are related to the Checkpoint Resume subsystem. Line 12 waits for callbacks to be enqueued.

Lines 21–42 communicate with the detector thread. They are almost identical to the `rcu_synchronize_shared()` function shown in listing 5.5. Lines 27–34 are related to the CPR subsystem and described in subsection 5.1.12.

Listing 5.10: The RCU reclaimer thread

```
1 mutex_enter(&RCU_CPU(cp, rcu_cpr_mutex));
2 /* ... */
3 for (;;) {
	if (!(RCU_CPU(cp, rcu_nextlist) || RCU_CPU(cp, rcu_curlist))) {
		do {
			if (RCU_CPU(cp, rcu_sync))
				rcu_handle_sync(cp);
			if (RCU_STOP == RCU_CPU(cp, rcu_thread_state))
				goto reclaimer_terminate;
			CALLB_CPR_SAFE_BEGIN(&cprinfo);
			mutex_exit(&RCU_CPU(cp, rcu_cpr_mutex));
			sema_p(&RCU_CPU(cp, rcu_queue_sema));
			mutex_enter(&RCU_CPU(cp, rcu_cpr_mutex));
			CALLB_CPR_SAFE_END(&cprinfo,
			    &RCU_CPU(cp, rcu_cpr_mutex));
			} while (!RCU_CPU(cp, rcu_nextlist));
	} else {
		membar_consumer();
	}

	gp_ctr = rcu_cb.rcu_gp_ctr;

	mutex_enter(&rcu_cb.rcu_gp_mutex);
	if (rcu_cb.rcu_gp_ctr - gp_ctr >= 2) {
		goto elapsed;
	}
	while (*rcu_cb.rcu_cpr_events & CALLB_CPR_START) {
		mutex_exit(&rcu_cb.rcu_gp_mutex);
		CALLB_CPR_SAFE_BEGIN(&cprinfo);
		delay(rcu_gp_max);
		CALLB_CPR_SAFE_END(&cprinfo,
		    &RCU_CPU(cp, rcu_cpr_mutex));
		mutex_enter(&rcu_cb.rcu_gp_mutex);
	}
	if (RCU_CPU(cp, rcu_weight) > rcu_weight_max) {
		++rcu_cb.rcu_hiwater;
		cv_signal(&rcu_cb.rcu_hiwater_cond);
		rcu_synchronize_common(1);
		--rcu_cb.rcu_hiwater;
	} else {
		rcu_synchronize_common(1);
	}
elapsed:
	mutex_exit(&rcu_cb.rcu_gp_mutex);
	rcu_advance_callbacks(cp);
}
```

The function for advancing callbacks is shown in listing 5.11. It first attempts to enqueue or execute callbacks in the same order as they appear on the list. The `taskq_dispatch()` operation may fail. If this happens, all the nonblocking callbacks are handled expediently, skipping the blocking ones. Finally, the skipped blocking callbacks are enqueued using a blocking invocation of `taskq_dispatch()`, which may sleep, waiting for sufficient resources to become available. (Such a situation should occur rarely. It can be caused by memory pressure or by a flood of sleepable callbacks. The second case can happen when the task queue is heavily contended, as already described in subsection 4.3.2.)

Atomic operations are used so that interrupt handlers interrupting the reclaimer thread and producing callbacks concurrently can not cause any problems.

Lines 6–66 handle all the callbacks waiting for the current grace period boundary, as described above. The weight values are taken into account and subtracted from the per CPU weight sum. Lines 68–74 advance the *nextlist* callbacks by moving them into the *curlist*. Note that the *curlist* will not be terminated correctly after this function exits. In fact it would be unsafe to terminate it, since assigning the trailing `NULL` pointer could race with a thread still appending to the former *nextlist*. Terminating the list is postponed until the next iteration and occurs on line 14.

The `RCU_VOL` macro generates volatile casts necessary to prevent the compiler from local variable elimination and code reordering. Details of these possible optimizations that could clobber the shared data in SMP environments have been described by Paul McKenney in his paper on preemptible RCU for Linux. [10]

Listing 5.11: Advancing callbacks

```
 1 static void
 2 rcu_advance_callbacks(cpu_t *cp)
 3 {
 4         rcu_t   *list = RCU_CPU(cp, rcu_curlist);
 5
 6         if (list) {
 7                 taskq_t         *taskq;
 8                 rcu_t           *next;
 9                 rcu_t           *failed_list = NULL;
10                 rcu_t           **failed_end = &failed_list;
11                 uint32_t        weight_sum = 0;
12
13                 taskq = RCU_VOL(taskq_t *, rcu_cb.rcu_taskq);
14                 *RCU_CPU(cp, rcu_curtail) = NULL;
15                 do {
16                         next = list->rcu_next;
17                         weight_sum += list->rcu_weight;
18
19                         switch (list->rcu_type) {
20                         case RCU_CB_DEFAULT:
21                                 list->rcu_func(list);
22                                 break;
23                         case RCU_CB_EXCL:
24                                 if (!taskq_dispatch(taskq,
25                                     (task_func_t *)list->rcu_func, list,
26                                     TQ_NOSLEEP | TQ_NOQUEUE)) {
27                                         *failed_end = list;
28                                         failed_end = &list->rcu_next;
29                                         list = next;
30                                         goto failure;
31                                 }
32                                 break;
33                         default:
```

```
                                panic("rcu_reclaimer:␣unknown␣callback␣type");
                                break;
                        }
                } while ((list = next) != NULL);

                goto out;
failure:
                for (; list; list = next) {
                        next = list->rcu_next;
                        weight_sum += list->rcu_weight;

                        switch (list->rcu_type) {
                        case RCU_CB_DEFAULT:
                                list->rcu_func(list);
                                break;
                        case RCU_CB_EXCL:
                                *failed_end = list;
                                failed_end = &list->rcu_next;
                                break;
                        }
                }

                *failed_end = NULL;
                do {
                        next = failed_list->rcu_next;

                        (void) taskq_dispatch(taskq,
                            (task_func_t *)failed_list->rcu_func,
                            failed_list, TQ_SLEEP);
                } while ((failed_list = next) != NULL);
out:
                RCU_SUB(RCU_CPU(cp, rcu_weight), weight_sum);
        }

        RCU_CPU(cp, rcu_curlist) = RCU_VOL(rcu_t *, RCU_CPU(cp, rcu_nextlist));
        if (RCU_CPU(cp, rcu_curlist)) {
                RCU_VOL(rcu_t *, RCU_CPU(cp, rcu_nextlist)) = NULL;
                RCU_VOL(rcu_t **, RCU_CPU(cp, rcu_curtail)) =
                    atomic_swap_ptr(&RCU_CPU(cp, rcu_nexttail),
                        &RCU_CPU(cp, rcu_nextlist));
        }
}
```

5.1.9 Waiting for Callbacks to Finish

As already mentioned, line 6 in listing 5.10 checks whether a request to announce the end of callback processing has been placed. This functionality can be accessed using the `rcu_call_synchronize()` API function. This function blocks until all previously enqueued callbacks have been processed.

The function consists of two calls, as shown in listing 5.12. First of all, reclaimer threads are all asked to announce an idle state as soon as all their callback lists become empty. This happens on line 4. (Waiting for the announcements might take an unbounded amount of time on a system constantly loaded with RCU callback processing. Improving the algorithm to achieve a reasonably fast response on extremely loaded systems is future work.)

Once all the reclaimer threads have gone through an idle state with empty callback lists, one can be sure that all the previously enqueued standard callbacks have been processed and all the *exclusive* callbacks have been dispatched. Line 5 waits for all the dispatched *exclusive* callbacks to finish. This functionality is provided by the task queues in UTS.

Listing 5.12: Waiting for callbacks to finish

```
1 void
2 rcu_call_synchronize()
3 {
4         rcu_call_synchronize_common();
5         taskq_wait(rcu_cb.rcu_taskq);
6 }
```

Listing 5.13 shows how reclaimers are asked to announce an idle state and how the announcements are waited for. Lines 12–15 take a short path if the thread was blocking on the mutex for a sufficiently long period of time. Blocking for the duration of two counter flips guarantees that all the callbacks produced before the call must have been already processed. A pointer to a local counter is set in all the per CPU data structures and all the reclaimer threads are notified by incrementing their queue semaphores. This is what lines 16–24 do. The semaphore increment wakes up the reclaimers in case they were sleeping. Lines 27–28 wait until all the reclaimer threads announce an idle state by decrementing the counter. Since the counter was set to the number of online CPUs, all the reclaimers will have gone through an idle state as soon as the counter reaches zero.

There are many technical subtleties regarding the behavior of the algorithm in cases when concurrent CPU-related events happen. Curious readers can read the source code comments around the `rcu_cpu_event()` function.

Listing 5.13: Asking reclaimers for an idle state

```
 1 static void
 2 rcu_call_synchronize_common()
 3 {
 4         uint32_t        counter;
 5         cpu_t           *cp;
 6         gp_t            sync_ctr;
 7
 8         membar_full();
 9         sync_ctr = rcu_cb.rcu_sync_ctr;
10
11         mutex_enter(&rcu_cb.rcu_sync_mutex0);
12         if (rcu_cb.rcu_sync_ctr - sync_ctr >= 2) {
13                 mutex_exit(&rcu_cb.rcu_sync_mutex0);
14                 return;
15         }
```

```
        mutex_enter(&cpu_lock);
        counter = ncpus_online;
        membar_producer();
        cp = cpu_active;
        do {
                RCU_CPU(cp, rcu_sync) = &counter;
                sema_v(&RCU_CPU(cp, rcu_queue_sema));
        } while ((cp = cp->cpu_next_onln) != cpu_active);
        mutex_exit(&cpu_lock);

        mutex_enter(&rcu_cb.rcu_sync_mutex1);
        while (counter)
                cv_wait(&rcu_cb.rcu_sync_cond, &rcu_cb.rcu_sync_mutex1);
        mutex_exit(&rcu_cb.rcu_sync_mutex1);
        ++rcu_cb.rcu_sync_ctr;
        mutex_exit(&rcu_cb.rcu_sync_mutex0);
}
```

Listing 5.14 shows how the reclaimer threads react to an idle state request. (The function is called in listing 5.10 on line 7.) First of all, line 7 invalidates the per CPU counter pointer and line 8 forces this change to occur before subsequent memory stores. The memory barrier is a price that must be paid for keeping the function lock-free for all reclaimer threads but one. Only one of the reclaimers actually acquires the mutex and notifies the waiting thread.

Line 9 contains an atomic operation that decrements the shared counter. This implementation speeds up the fast path (decrementing the counter to a nonzero value) at the expense of possible spurious wake-ups of the waiting thread, a memory barrier and an explicit atomic instruction. The spurious wake-up events are benign, as discussed in the source code comments. The lock-free fast path avoids lock contention by requiring only one reclaimer per idle state request to acquire the mutex, no matter how many processors the system may have.

Listing 5.14: Handling an idle state request

```
 1 static void
 2 rcu_handle_sync(cpu_t *cp)
 3 {
 4         uint32_t        *rcu_sync;
 5
 6         rcu_sync = RCU_CPU(cp, rcu_sync);
 7         RCU_CPU(cp, rcu_sync) = NULL;
 8         membar_producer();
 9         if (!atomic_dec_32_nv(rcu_sync)) {
10                 mutex_enter(&rcu_cb.rcu_sync_mutex1);
11                 cv_signal(&rcu_cb.rcu_sync_cond);
12                 mutex_exit(&rcu_cb.rcu_sync_mutex1);
13         }
14 }
```

5.1.10 The Detector Algorithms

When there is no activity related to RCU, the grace period detector thread sleeps. Once some of the RCU reclaimers or other threads want to observe a grace period boundary, they have to wake up the detector and wait for its announcement.

The detector first takes a snapshot of some CPU related data, such as the last observed grace period counter and the context switch counter. It excludes processors that will not need attention, such as the current processor or those that have already observed the current grace period counter state. Then, if *high water* is not announced, the detector sleeps for a couple of clock ticks.

After waking up, the detector rescans the remaining processors, verifying that they have already become aware of the current grace period. If some of them remained in an unknown state, the detector thread has to force a context switch on those processors to make sure any read-side critical sections running on them have terminated. (Since all read-side sections run with disabled preemption, context-switching the detector to the CPUs guarantees that all of them have ended.)

5.1.11 Detector Code Walkthrough

The detector thread's main loop is shown in listing 5.15. Unlike the reclaimer threads, the detector never exits.

Obviously, a grace period boundary will be announced without being actually observed right after the detector wakes up. This is perfectly legal, based on the following two facts. First, the detector always observes one extra full grace period boundary before going to sleep. Second, callbacks are only processed after two grace period boundaries are announced (which can not happen without observing a full grace period) and threads waiting for grace periods to elapse also require two boundaries to be announced.

Listing 5.15: Grace period detector's main loop

```
mutex_enter(&rcu_cb.rcu_gp_mutex);
/* ... */
for (;;) {
        while (!rcu_cb.rcu_gp_wanted) {
                CALLB_CPR_SAFE_BEGIN(&cprinfo);
                cv_wait(&rcu_cb.rcu_wanted_cond, &rcu_cb.rcu_gp_mutex);
                CALLB_CPR_SAFE_END(&cprinfo, &rcu_cb.rcu_gp_mutex);
        }

        --rcu_cb.rcu_gp_wanted;
        ++rcu_cb.rcu_gp_ctr;
        cv_broadcast(
            &rcu_cb.rcu_elapsed_cond[rcu_cb.rcu_gp_ctr & 0x1]);

        mutex_exit(&rcu_cb.rcu_gp_mutex);
        rcu_wait_and_force();
        mutex_enter(&rcu_cb.rcu_gp_mutex);
}
```

Listing 5.16 shows the algorithm that detects processors requiring attention (lines 11–23) and waits for a bounded period of time (lines 30–40) in case if a *high water* state is not reported. Last, it forces some of the processors into a context switch when necessary (lines 44–54). The processor running the current thread, processors running userspace code and processors that have already observed the current grace period counter are excluded (lines 15–17). They will not need attention during this boundary detection.

After the waiting phase, some processor-related data must be rechecked, since releasing and reacquiring the `cpu_lock` mutex opens a race window in which processors could be added or removed. As `cpu_t` structures are never freed, the list of processors can still be safely traversed. Only the `cpu_next` pointer and the `CPU_ONLINE` flag have to be reexamined to make sure the CPU has not been removed or set offline (line 46).

New CPUs that were added during the waiting phase do not need any attention, since all the possible read-side sections running on them obviously started no sooner than after the preceding grace period boundary, which means they need not be waited for. Of course, the new processors will need (and receive) attention once a subsequent grace period boundary is detected.

Lines 12 and 13 do a simple housekeeping. Processors whose observed grace period counters are too far from the current counter value have their observed counters reinitialized. This may be necessary for processors that spent a considerable amount of time offline. The global grace period counter may have overflowed during that time. This assignment makes sure such a situation is always benign. (Else it could be harmful in one case out of 2^{32}. Failures would be highly improbable, but theoretically possible.)

Listing 5.16: Observing a grace period

```
 1 static void
 2 rcu_wait_and_force()
 3 {
 4         cpu_t   *cp, *first;
 5         cpu_t   *list_first;
 6         cpu_t   **list_tail = &list_first;
 7
 8         mutex_enter(&cpu_lock);
 9         first = CPU;
10         cp = first;
11         do {
12                 if (RCU_GP_GT(RCU_CPU(cp, rcu_gp_ctr), rcu_cb.rcu_gp_ctr))
13                         RCU_CPU(cp, rcu_gp_ctr) = rcu_cb.rcu_gp_ctr - 1;
14                 if (
15                         rcu_cb.rcu_gp_ctr != RCU_CPU(cp, rcu_gp_ctr) &&
16                         cp->cpu_mstate == CMS_SYSTEM &&
17                         cp->cpu_thread != curthread
18                 ) {
19                         RCU_CPU(cp, rcu_pswitch) = CPU_STATS(cp, sys.pswitch);
20                         *list_tail = cp;
21                         list_tail = &RCU_CPU(cp, rcu_next_cpu);
22                 }
23         } while ((cp = cp->cpu_next_onln) != first);
24         *list_tail = NULL;
25
```

```
        if (rcu_cb.rcu_hiwater) {
                membar_consumer();
        } else {
                mutex_exit(&cpu_lock);
                mutex_enter(&rcu_cb.rcu_gp_mutex);
                if (!rcu_cb.rcu_hiwater) {
                        clock_t timeleft = rcu_gp_max;
                        do {
                                timeleft = cv_reltimedwait(
                                    &rcu_cb.rcu_hiwater_cond,
                                    &rcu_cb.rcu_gp_mutex, timeleft,
                                    TR_MILLISEC);
                        } while (!(-1 == timeleft || rcu_cb.rcu_hiwater));
                }
                mutex_exit(&rcu_cb.rcu_gp_mutex);
                mutex_enter(&cpu_lock);
        }

        for (cp = list_first; cp; cp = RCU_CPU(cp, rcu_next_cpu)) {
                if (
                    cp->cpu_next && cpu_is_online(cp) &&
                    cp->cpu_mstate == CMS_SYSTEM &&
                    rcu_cb.rcu_gp_ctr != RCU_CPU(cp, rcu_gp_ctr) &&
                    RCU_CPU(cp, rcu_pswitch) == CPU_STATS(cp, sys.pswitch)
                ) {
                        affinity_set(cp->cpu_id);
                        affinity_clear();
                }
        }
        mutex_exit(&cpu_lock);
}
```

5.1.12 Kernel Integration: Interfacing with CPR

The Checkpoint/Resume subsystem provides a possibility to quiesce a running system and stop all threads in a consistent state. This has to be done before the machine can be switched to a low-power state, migrated to another physical host and the like. All threads are expected to either terminate or reach a CPR-safe state when a checkpoint is requested.

The detector thread announces a CPR-safe state when it sleeps, waiting for grace period requests to come. The situation is slightly more complicated for the reclaimer threads. They announce the safe state each time they sleep on their per CPU semaphores, waiting for callbacks to emerge. However, this is not sufficient to prevent checkpoint failures caused by RCU. When the reclaimer is waiting for an announcement from the detector, it could wait forever in case if the detector had been asked to reach a CPR-safe state and reached it. Consequently, the waiting reclaimer would not reach a CPR-safe state and cause the checkpoint request to fail.

As far as the author knows, the current CPR code does not include a mechanism that could construct a dependency DAG of threads and request safe states in the correct order. This is why an RCU-specific solution had to be found. Before waiting for the detector, reclaimers verify (under a mutex) whether the detector was asked to quiesce itself. If so, reclaimers suspend themselves for a standard grace period duration, in anticipation of receiving a CPR request as well. Before waiting, they announce a CPR safe state. They repeat this process as long as needed, which can be seen in listing 5.10 on lines 27–34.

It is possible that the reclaimer threads could be marked as CPR-safe, in which case they would not have to reach a CPR-safe state. The current approach, however, guarantees that a batch of callbacks (a set of callbacks waiting for the same grace period boundary) will not be split across a low power state.

Testing the cooperation of RCU with CPR is future work. This would require access to the standard unit tests used by Oracle, so that all the aspects of the required behavior could be verified.

5.1.13 Kernel Integration: Interfacing with CPU Hotplug

The RCU implementation exports a `rcu_cpu_event()` function, which is used as a callback handler reacting to various CPU-related events. This function initializes the RCU data stored in `cpu_t` structures and performs the necessary maintenance when new processors are detected (on boot or due to a hotplug event) and when processors are onlined and offlined. Specifically, a reclaimer thread must be spawned for each new processor and bound to that processor. If the number of available processors changes substantially (by a binary order of magnitude), the system-wide RCU task queue is reallocated to reflect the change.

Testing the reaction to CPU onlining and offlining thoroughly is future work. There are still some open questions, as described in section 8.1. It is necessary to choose a solution to the problems concerning offline processors that can handle interrupts. Once a solution is chosen, the code dealing with some of the CPU-related events might be rewritten or become obsolete. In either case, unit tests used by OS/Net developers would be very helpful, if not necessary.

5.1.14 Kernel Integration: The Big Image

The RCU algorithm is closely bound to the core kernel data structures and stores its data directly in the `cpu_t` data structure. It must be usable early during boot, even during the uniprocessor stage. Callbacks can be produced almost from the very beginning of the kernel's `main()` function. Threads related to RCU (the detector and reclaimers) are spawned as soon as the cache for `proc_t` structures becomes available.

For the reasons mentioned above, the author decided not to even attempt isolating the global RCU implementation into a loadable kernel module. Instead, the RCU implementation is linked with the `genunix` module. Since RCU causes precisely zero overhead when not used, this should not be an issue.

The RCU source code can be found in `uts/common/os/rcu.c`, `uts/common/sys/rcu.h` and `uts/common/sys/rcu_impl.h`.

5.1.15 Comparison of the UTS RCU to the Linux RCU

The current Linux RCU algorithm is much more scalable than the prototype implementation in UTS. Details about the hierarchical tree-based RCU algorithm used on Linux are described in subsection 2.1.4.

The Linux RCU provides multiple variants of the RCU algorithm, such as preemtible readers, and other advanced features. It has to cope with (or possibly take advantage of) some Linux specifics, such as optional kernel preemption and numerous other compile-time options.

As long as RCU readers behave responsibly, the UTS prototype implementation can provide an upper bound on grace period duration on non-real-time systems.

The prototype RCU implementation in UTS **does not depend on clock tick processing** in any way. This is probably the main difference. There are two main reasons for this design decision:

- All modern kernels are heading towards a tickless design. Idle states on Linux are already tickless and there are ongoing efforts to eliminate even regular scheduler ticks under load, making the operating system similar to a discrete simulation algorithm. Algorithms strongly based on clock ticks are not future proof and have to cope with CPU idle states.

- The UTS kernel uses a completely different clock tick handling mechanism than Linux does. On Linux, all CPUs receive clock interrupts when they are not idle. This is not the case on UTS. On Solaris 10 (as described in Solaris Internals [7]), there used to be only one processor handling clock ticks. The current OpenSolaris implementation has multithreaded clock tick processing, so that data locality and scalability requirements are met. However, this does not mean that clock ticks are processed by all CPUs. In fact the number of CPUs processing clock ticks is much lower than the total number of CPUs.

  Instead of instrumenting the clock tick thread with code run on every single tick, it seems better to provide a regular thread with a high priority instead. This is the detector thread. Its priority is such that it can preempt most other threads, with the exception of interrupt threads and real-time threads. The most importantly, it can sleep (causing exactly zero overhead) when no RCU-related activity is observed. The possibility of replacing the RCU detector and reclaimer threads with execution entities closer to interrupt threads and possible benefits of this approach are discussed in chapter 8.

The prototype RCU implementation in UTS **does not instrument or modify the scheduler code** in any way and does not require any explicit actions to be taken on every context switch. Instrumenting the scheduler code in UTS does not make sense:

- The UTS kernel can (under certain circumstances) schedule huge time quanta (more than a second) to low-priority threads in the Fair-Share scheduling class. [7] This implies that RCU needs to use different techniques of limiting the grace period duration.

5.2 The QRCU Implementation

The QRCU algorithm has been ported from Linux, with all the necessary changes related to the different semantics of some synchronization primitives on both kernels. The original version of the algorithm was designed by Oleg Nesterov in 2007. Paul McKenney improved the algorithm and added a lockless fast path. He then prove the correctness of his approach both formally and using the Promela language. Both proofs of correctness and other details can be found in [9].

5.2.1 Read-Side Algorithm

Readers increment an atomic variable based on the state of a global counter, as long as the atomic variable is not zero. In the latter case, the reader has to re-read the global counter and retry the increment with its new value, possibly referencing a different atomic variable. At the end, the reader decrements the same atomic variable it had originally incremented. Further details about this algorithm are mentioned in subsection 2.2.4.

5.2.2 Read-Side Code Walkthrough

The read-side locking function is displayed in listing 5.17. It uses the *increment if not zero* primitive, which had to be ported from the Linux kernel together with the QRCU algorithm. (More precisely, QRCU has never been accepted into the mainline kernel, but can be obtained from a patch against the 2.6.19 version.) The Linux kernel contains numerous „do if not something“ atomic operations, but this project did not attempt to port more of them and provide a library. Instead, only the needed one, shown in listing 5.18, was included directly into the QRCU source code.

Listing 5.17: Starting a QRCU read-side critical section

```
int
qrcu_read_lock(qrcu_t *qrcu)
{
        for (;;) {
                int     idx = QRCU_VOL(uint32_t, qrcu->qrcu_ctr) & 0x1;
                if (qrcu_inc_nz(&qrcu->qrcu_rlock[idx]))
                        return idx;
        }
}
```

Listing 5.18: Incrementing a nonzero counter

```
static inline uint32_t
qrcu_inc_nz(uint32_t *atomic)
{
        int old;
        int aux = QRCU_VOL(uint32_t, *atomic);

        while (aux && aux != (old = atomic_cas_32(atomic, aux, aux + 1)))
                aux = old;
        return aux;
}
```

The read-side critical section epilogue can be seen in listing 5.19. The mutex used to notify the waiting reclaimer should be initialized as a spinlock in cases when readers are required to be non-blocking. The current implementation uses a standard adaptive mutex for the sake of simplicity. Presumably, when readers at high interrupt levels are expected, initializing `qrcu_mutex1` as a spinlock is the only viable solution.

Listing 5.19: Ending a QRCU read-side critical section

```
void
qrcu_read_unlock(qrcu_t *qrcu, int idx)
{
        if (!atomic_dec_32_nv(&qrcu->qrcu_rlock[idx])) {
                mutex_enter(&qrcu->qrcu_mutex1);
                cv_signal(&qrcu->qrcu_cond);
                mutex_exit(&qrcu->qrcu_mutex1);
        }
}
```

5.2.3 Write-Side Algorithm

Readers interested in the intricacies of the write-side algorithm and the proof of its correctness are encouraged to read the whole paper by Paul McKenney. [9]

The fast path algorithm is designed so that a writer will either avoid racing with both readers and other writers, or race with other writers in a precisely defined manner. (The former implies that all threads are in a quiescent state. The latter implies that at least two grace period boundaries have elapsed.) In both cases, the reclaimer can proceed without acquiring the mutex and without interacting with other threads.

The slow path first tries the standard optimization seen in listing 5.4, lines 10–13. Blocking for at least two counter flips (or grace period boundaries in the standard RCU parlance) guarantees that a full grace period must have elapsed. In such cases, it is not necessary to increment the global counter or wait.

When all the optimizations above fail, the counters are swapped, as described in subsection 2.2.4, and the writer has to wait for the old counter to become zero. Once this happens, the writer becomes a reclaimer and can be sure no readers can access the old data it is about to remove.

5.2.4 Write-Side Code Walkthrough

The code separating writers from reclaimers is shown in listing 5.20. Line 6 forces ordering of the preceding memory operations before the subsequent fastpath tests. Lines 7–15 implement Paul McKenney's fast path. Line 10 forces load ordering so that the two tests of counters are independent. Lines 19–23 check whether the thread has blocked long enough for a full grace period to elapse.

The slow path flips the counter on lines 25–28. The new counter must be incremented before decrementing the old one. This guarantees that the new counter will not be decremented to zero during the grace period to follow, until a subsequent reclaimer flips the counter again. Without this assumption, spurious wake-ups could occur and in some very special cases (of aggressively reordering CPUs), data consistency could be compromised.

Last but not least, lines 31–32 wait for all the readers to terminate. Once the counter reaches zero, no more readers will be able to increment it. This implies that subsequent readers will eventually notice that the counter has been flipped (possibly by retrying the counter index computation) and preceding readers have already finished. A grace period has ended.

Listing 5.20: Waiting for the QRCU readers to finish

```
 1 void
 2 qrcu_synchronize(qrcu_t *qrcu)
 3 {
 4         int idx;
 5
 6         membar_full();
 7         if (QRCU_VOL(uint32_t, qrcu->qrcu_rlock[0]) +
 8             QRCU_VOL(uint32_t, qrcu->qrcu_rlock[1]) <= 1) {
 9                 membar_consumer();
10                 if (QRCU_VOL(uint32_t, qrcu->qrcu_rlock[0]) +
11                     QRCU_VOL(uint32_t, qrcu->qrcu_rlock[1]) <= 1) {
12                         membar_full();
13                         return;
14                 }
15         }
16
17         mutex_enter(&qrcu->qrcu_mutex0);
18         idx = qrcu->qrcu_ctr & 0x1;
19         if (QRCU_VOL(uint32_t, qrcu->qrcu_rlock[idx]) == 1) {
20                 mutex_exit(&qrcu->qrcu_mutex0);
21                 membar_enter();
22                 return;
23         }
24
25         atomic_inc_32(&qrcu->qrcu_rlock[idx ^ 0x1]);
26         membar_producer();
27         ++qrcu->qrcu_ctr;
28         atomic_dec_32(&qrcu->qrcu_rlock[idx]);
29
30         mutex_enter(&qrcu->qrcu_mutex1);
31         while (qrcu->qrcu_rlock[idx])
32                 cv_wait(&qrcu->qrcu_cond, &qrcu->qrcu_mutex1);
33         mutex_exit(&qrcu->qrcu_mutex1);
34         mutex_exit(&qrcu->qrcu_mutex0);
35
36         membar_enter();
37 }
```

5.2.5 Kernel Integration

The QRCU algorithm is implemented as a loadable kernel module called `misc/qrcu`.

If it is ever needed early on boot, it could either become part of the core kernel (the `genunix` module) or be loaded on demand using the `modstubs.s` mechanism. (The latter solution is used for scheduling classes and some other modules known in advance at compile time. The kernel implements function stubs with names identical to the interface provided by a loadable module. Invocation of one of the stubs causes the corresponding skeleton module to get loaded and service the request.)

The source code of QRCU can be found in `uts/common/qrcu/qrcu.c` and `uts/common/sys/qrcu.h`.

5.3 The DRCU Implementation

DRCU stands for „Dummy RCU“ and implements the core part of the RCU API using a readers-writer lock. This algorithm has been implemented for benchmarking purposes only. It is prone to various deadlock scenarios if invoked from interrupt context. It also suffers from severe performance issues and does not guarantee readers to never block. It documents the relationship between RCU and other types of locking and was one of the „toy“ RCU implementations originally presented by Paul McKenney. [12]

5.3.1 DRCU Code Walkthrough

Listings 5.21 and 5.22 show the read-side primitives. Listing 5.23 shows the blocking API. This implementation does not provide a callback API. As described in the 2.2.1 subsection, the algorithm is interesting from the theoretical point of view rather than for practical qualities. As observed in chapter 7, it has a catastrophic SMP performance, problematic even with as few as eight processors.

Listing 5.21: Starting a DRCU read-side critical section

```
void
drcu_read_lock(drcu_t *drcu)
{
        rw_enter(&drcu->drcu_rwlock, RW_READER);
}
```

Listing 5.22: Ending a DRCU read-side critical section

```
void
drcu_read_unlock(drcu_t *drcu)
{
        rw_exit(&drcu->drcu_rwlock);
}
```

Listing 5.23: Waiting for DRCU readers to finish

```
void
drcu_synchronize(drcu_t *drcu)
```

```
{
        rw_enter(&drcu->drcu_rwlock, RW_WRITER);
        rw_exit(&drcu->drcu_rwlock);
}
```

5.3.2 Kernel Integration

The DRCU algorithm is implemented as a loadable kernel module called `misc/drcu`. The source code of DRCU can be found in `uts/common/drcu/drcu.c` and `uts/common/sys/drcu.h`.

5.4 Summary of RCU Algorithms

Table 5.1 summarizes the features, characteristics and limitations of the presented RCU algorithms. Some of the boolean values are commented in more detail.

	DRCU	QRCU	RCU
readers can sleep	YES (1)	YES	NO
readers in interrupt contexts	NO	YES	YES (2)
blocking API	YES	YES	YES
callback API	NO	NO	YES
callbacks can sleep	—	—	YES (3)
callbacks from interrupt contexts	—	—	YES
bounded grace period	NO	NO	YES (4)
local, multiple instances	YES	YES	NO

Table 5.1: Summary of RCU algorithms

1. This depends on the limitations of a readers-writer lock implementation on the particular platform. Many implementations do not allow readers to block or sleep when holding the lock.

2. Although this RCU algorithm is designed to support all of its non-blocking operations in interrupt contexts, there are technical difficulties that have to be dealt with first. Section 8.1 summarizes the problem and possible solutions.

3. This feature is available and fully functional, but should be used with care and avoided when possible, for performance reasons. Sleepable callbacks are handled by a much less efficient mechanism than the non-blocking ones.

4. Grace periods are bounded as long as there is no extreme real-time or interrupt workload and all read-side sections take a bounded amount of time.

Chapter 6

Other Contributions to UTS

The prototype implementation of RCU for the UTS kernel, described in chapter 5, is the main contribution of this thesis. There are also other contributions related to benchmarking the RCU algorithms and an example application of the RCU API. This chapter describes the related work, focusing on a non-blocking hash table implementation based on RCU.

6.1 The Non-Blocking Hash Table

Paul McKenney describes a non-blocking hash table algorithm in his Ph. D. thesis. [3] McKenney's algorithm has been implemented from scratch by this author and works in the UTS kernel. According to McKenney, non-blocking hash tables are used in the K42 kernel. The new implementation in UTS has been written without reading the K42 sources and does not use any code from K42. The hash table supports the standard three hash map operations — mapping, unmapping and retrieval of data objects by keys. All the three operations are non-blocking.

All the table operations (and especially the non-blocking removals) are based on the notion of *existence locks*. [3] The table requires a mechanism that prevents data items from being removed in a moment when they are accessed. RCU provides such a mechanism. Performing all the operations inside RCU read-side critical sections guarantees that hash chain traversals will always see valid data. Threads can possibly access stale data, but they will never access invalid data, as long as they play by the rules.

The non-blocking nature of the hash table makes it possible to run read and insert operations concurrently with no synchronization at all. In these special cases, there is no need for RCU. However, when all the three possible operations (including removals) can run concurrently, all of them must be protected by RCU read-side critical sections to prevent stale hash chain items they might encounter from being reclaimed too early. Furthermore, threads that remove an item from a hash chain must wait for at least one grace period before they can safely reclaim the item. This can be done either synchronously, or using the RCU callback mechanism.

The hash table algorithm must guarantee that when multiple threads insert a key concurrently, at most one of them will succeed. More precisely, exactly one of them will succeed if and only if the hash table did not contain the inserted key yet. Similarly, when multiple threads remove a key concurrently, at most one of them can succeed. Exactly one will succeed if and only if the table contained the removed key. These rules are obeyed with the help of atomic instructions.

The standard non-blocking list manipulation algorithms had to be modified so that they can tolerate stale data, as described by McKenney. [3] New elements can only be added to the head of the hash chain. Successors of removed elements can only be connected to valid predecessors.

The insert operation first takes a snapshot of the head of the hash chain, then verifies whether the hash chain contains the new key. Finally, when no conflicting key is found, it connects the new element using an atomic compare and swap instruction. Adding new elements or removing the first element in the mean time will cause the atomic operation to fail and the hash chain will have to be traversed again. As long as the operation runs in an RCU read-side critical section, all the elements of the traversed hash chain are guaranteed to remain valid.

The remove operation first searches the hash chain to find the element with the requested key. If found, the element is invalidated atomically. This is done by setting the lowest-order bit of its `next` pointer to 1 using an atomic operation. This guarantees that each element can only be invalidated once. And even more importantly, a concurrent removal of the successor will either terminate before this atomic invalidation, or spin until the current removal has completed. This is guaranteed by using the the lowest order bit of the `next` pointer as the „stale flag“.

After invalidating the removed element atomically, the removal operation proceeds to assigning the `next` pointer of the current element to the predecessor's `next` pointer. Again, an atomic CAS operation is used. Since the CAS operation expects a valid pointer (with zero lowest-order bit) to be replaced, the successor will always be connected to a valid predecessor. If the CAS operation fails, the hash chain must be traversed again to find a new valid predecessor. This is repeated until either a valid predecessor is found and successfully connected to the successor, or the current element can not be found on the hash chain any more. The second case can happen as a result of racing with concurrent insert operations, as described in the code walkthrough below.

Since the hash table search operation is trivial, we will only give a brief description of how elements are inserted and removed. Listing 6.1 shows the insert operation. Line 8 takes a snapshot of the hash chain head in such a way that it is fetched from the memory only once. Otherwise the compiler could eliminate the `first` and `retfirst` variables and read the memory location multiple times. Obtaining `first` during or after the actual chain traversal, as seen by other CPUs, could result in a successful insertion of multiple elements with the same key.

The `ht_get_valid()` function on line 11 returns a reference to a valid item with a matching key into `prev` and stores a reference to the first valid item into `next`. The `next` pointer is then used to reduce contention by circumventing all invalid elements at the beginning of the list (if any). This avoids further racing with threads that are concurrently removing the leading invalidated elements, reducing the probability of starvation. [3]

Once the item is initialized properly, line 15 will make sure that all these changes will be visible before the item becomes accessible to other threads. The atomic operation on line 16 attempts to connect the item to the head of the list, using the snapshot kept in `first`. If this fails, someone has modified the hash chain head concurrently and the operation will have to be retried.

On success, null is returned. Else line 13 returns a reference to the conflicting item with the same key.

Theoretically, the algorithm could circumvent all the leading invalid items even when a conflicting valid item is found. However, the author has chosen not to take this approach in order to keep the code simple. Furthermore, it does not seem reasonable to race with concurrent insert and remove operations when the current insert operation is about to fail and does not add anything to the hash chain.

Listing 6.1: The non-blocking hash table *insert* operation

```
 1 ht_t *
 2 ht_add(ht_map_t *map, ht_key_t key, ht_t *value)
 3 {
 4         _ht_bucket_t    *bucket = HT_HASH(map, key);
 5         ht_t            *prev, *first, *retfirst, *next;
 6 
 7         value->ht_key = key;
 8         retfirst = HT_VOL(ht_t *, bucket->ht_first);
 9         do {
10                 first = retfirst;
11                 prev = ht_get_valid(first, key, &next);
12                 if (prev)
13                         return (prev);
14                 value->ht_next = next;
15                 membar_producer();
16         } while ((retfirst = atomic_cas_ptr(&bucket->ht_first, first, value)) !=
17             first);
18 
19         return (NULL);
20 }
```

The remove operation, displayed in listing 6.2, starts with a hash chain traversal on line 8. A reference to a valid item with a matching key is returned into `dead` and a reference to its predecessor is stored into `pred`. When no valid matching item is found, the function takes the short path.

Lines 12 and 13 read the original pointer and make sure the local copy is valid. The valid pointer is stored in `newnext` and its invalidated form (with the lowest-order bit set to 1) is stored in `newnext`. The atomic operation on line 14 attempts to change the valid pointer to its invalid counterpart atomically. If someone connected a different successor concurrently, the algorithm spins. If someone else invalidated the current item concurrently, the short path is taken and a failure is reported as if no matching item was found.

The memory barrier on line 22 guarantees that the invalidation will reach global visibility before the invalidated item is disconnected. The comments in the full source code describe why this is crucial for keeping the hash chain consistent. The atomic operation on line 24 tries to disconnect the current item from the hash chain by connecting its successor to a valid predecessor. If the predecessor was invalidated concurrently, line 25 repeats the lookup and finds a new predecessor. If no predecessor is found, then the current item is no longer a member of the hash chain. This can happen due to a race with an insert operation that circumvented leading invalid items, as already described. In such case, the removal operation succeeds, since the invalidation was successful, but it does not need to disconnect the item.

Listing 6.2: The non-blocking hash table *remove* operation

```
 1 ht_t *
 2 ht_del(ht_map_t *map, ht_key_t key)
 3 {
 4         _ht_bucket_t    *bucket = HT_HASH(map, key);
 5         ht_t            *dead, *oldnext, *newnext, *retnext;
 6         ht_t            **pred;
 7 
 8         dead = ht_get_pred(&bucket->ht_first, key, &pred);
 9         if (!dead)
10                 return (NULL);
11 
12         oldnext = HT_PTR(HT_VOL(ht_t *, dead->ht_next));
13         newnext = HT_INVAL_PTR(oldnext);
14         while ((retnext = atomic_cas_ptr(&dead->ht_next, oldnext, newnext)) !=
15             oldnext) {
16                 if (!HT_VALID_PTR(retnext))
17                         return (NULL);
18                 oldnext = retnext;
19                 newnext = HT_INVAL_PTR(retnext);
20         }
21 
22         membar_producer();
23 
24         while (atomic_cas_ptr(pred, dead, oldnext) != dead) {
25                 pred = ht_find_pred(&bucket->ht_first, dead);
26                 if (!pred)
27                         return (dead);
28         }
29         return (dead);
30 }
```

The hash table algorithm is implemented as a loadable kernel module. The module is called `misc/htab`. The source code can be found in `uts/common/htab/htab.c`, `uts/common/htab/htab_impl.h` and `uts/common/sys/htab.h`.

6.2 The SMP Barrier

For the sake of benchmarking, an SMP barrier primitive was needed to control multi-threaded benchmarks. The benchmarking code uses barriers extensively. Since no suitable implementation was found in the core UTS kernel, one has been created, based on mutexes and condition variables. Both testing and benchmarking modules described in chapter 7 depend on it.

The barrier implementation is provided in the form of a loadable kernel module called `misc/barrier`. The source code can be found in `uts/common/barrier/barrier.c` and `uts/common/os/barrier.h`.

6.3 RCU Code Examples and Tests

The RCU algorithms have been tested and benchmarked using the `misc/rcutest` and `misc/rcudemo` loadable kernel modules. The source code of these two modules can be found in `uts/common/rcutest/rcutest.c` and `uts/common/rcudemo/rcudemo.c`, respectively. The purpose and function of these modules are described in chapter 7.

The `rcutest` module contains sanity checks and stress tests for the non-blocking hash table insert operation and for the RCU implementations.

The `rcudemo` module provides an abstraction over the three different RCU APIs implemented in UTS and uses this abstraction to benchmark the RCU algorithms. The benchmarks simulate a real-life workload by using RCU to perform concurrent operations on a non-blocking hash table. All the inserted and retrieved data objects are read and overwritten multiple times to simulate standard memory operations and to verify the correctness of RCU at the same time.

The abstraction used by `rcudemo` to hide the differences between the three implementations of RCU is good for code reusability, but bad for readability. The `rcudemo` module could be hard to understand for readers unfamiliar with RCU. To overcome this inconvenience, a complete set of code examples demonstrating the cooperation of RCU with the non-blocking hash table can be found in appendix A.

Chapter 7

Testing and Benchmarking RCU in UTS

All the three RCU algorithms were tested and benchmarked using specialized kernel modules described in this chapter. The benchmarking algorithm is described here as well. Finally, results of the benchmark are listed. Three different multiprocessor machines were used for the experiments.

7.1 Testing RCU and the Hash Table

The `misc/rcutest` kernel module provides basic sanity checks of all the three RCU algorithms. It also tests the insertion algorithm of the non-blocking hash table. The module spawns a kernel process, which performs the following tests in a sequence:

- `3 * ncpus` threads are spawned, all of them producing millions of callbacks for the global RCU. The callback functions decrement counters atomically. This verifies that the callback framework works and all callbacks are eventually executed. One callback in 256 is enqueued as `exclusive`, which means that the reclaimer threads will dispatch it into a task queue instead of executing it directly. (Stressing the task queue more intensively might expose the contention issues mentioned in subsection 4.3.2.)
- `3 * ncpus` threads are spawned, all of them inserting items into a shared non-blocking hash table. The maximum average chain length is approximately 16 (but can slightly vary on machines where the number of processors is not a power of 2). The ranges of inserted keys overlap, so that all the threads have an opportunity to test both successful and unsuccessful insertions and race with each other. The test is repeated a number of times and consistency of the table is verified after each iteration.
- For each RCU implementation, `3 * ncpus` readers and one writer are spawned. This is often called the *dual buffer test* in papers related to RCU. The writer

swaps (pointers to) two buffers repeatedly, waiting for a grace period after each swap operation and rewriting the buffer twice afterwards, verifying its contents before each rewrite. Readers perform exactly the same procedure when accessing the buffer under a read-side critical section. They overwrite it twice, verifying that values they read could only result from racing with other readers, not the writer. (Presumably, the writer expects and verifies that it has exclusive access to the buffer after waiting for a grace period.) Four different buffer sizes are tested (512 kB, 32 kB, 2 kB and 128 B) for each RCU algorithm and millions of reader iterations are performed.

The testing module requires approximately 2 gigabytes of available physical memory per 8 processors. The number of test iterations and other constants have to be tweaked on machines with a high number of processors and insufficient amount of memory.

The code used for buffer consistency checking is almost identical for both the `misc/rcutest` module and the `misc/rcudemo` module (described in the following section 7.2). The `rcudemo` module omits a memory barrier that `rcutest` uses to separate the two buffer overwrites (and make race conditions more probable), but this is the only difference. Since `rcudemo` is designed to simulate a standard workload with common operations on RCU protected data, it does not use memory barriers and other expensive operations explicitly.

Listing 7.1 shows how readers access the buffer they obtain. During the first iteration (lines 9–21), the reader could be accessing a buffer not yet touched by other readers, in which case the writer's second (`WRITER2`) pattern would be observed. Otherwise this reader could be racing with other readers, in which case some of the two readers' patterns can be seen. Observing the writer's first (`WRITER1`) pattern is a bug which has to be reported immediately. Similarly, observing any of the writer's patterns during the second iteration (lines 23–34) is a bug, since only readers should be allowed to access the buffer and the current reader has already overwritten any possible writer's patterns.

The return value of the function is used to compute statistics on how often stale data was observed before the two iterations (*early stale data*) and thereafter (*late stale data*). Ideally, fresh data should be observed at least an order of magnitude more frequently than late stale data. Furthermore, early stale data is expected to occur less frequently than late stale data. The reader threads print out stale data statistics when they finish.

Listing 7.1: Reader's buffer access

```
 1 static uint32_t
 2 check_db_reader_verify(buffer_t *buf)
 3 {
 4         const uint32_t  *const end = buf->b_data + buf->b_size;
 5         uint32_t        *cur;
 6         uint32_t        stale1 = rcu_access(uint32_t, buf->b_stale);
 7         uint32_t        stale2;
 8 
 9         for (cur = buf->b_data; cur < end; ++cur) {
10                 switch (*cur) {
11                 case WRITER2: /* FALLTHROUGH */
12                 case READER1: /* FALLTHORUGH */
13                 case READER2:
14                         break;
15                 default:
16                         panic("\tReader␣error␣at␣%ld:␣Found␣pattern␣%x.\n",
17                             (long)(cur - buf->b_data), *cur);
18                         break;
19                 }
20                 *cur = READER1;
21         }
22         membar_full();
23         for (cur = buf->b_data; cur < end; ++cur) {
24                 switch (*cur) {
25                 case READER1: /* FALLTHROUGH */
26                 case READER2:
27                         break;
28                 default:
29                         panic("\tReader␣error␣at␣%ld:␣Found␣pattern␣%x.\n",
30                             (long)(cur - buf->b_data), *cur);
31                         break;
32                 }
33                 *cur = READER2;
34         }
35 
36         stale2 = rcu_access(uint32_t, buf->b_stale);
37         if (stale1 && !stale2) {
38                 panic("\tReader␣error:␣Stale␣data␣became␣fresh.\n");
39         }
40 
41         return (stale1 + stale2);
42 }
```

Listing 7.2 shows how writers access the buffer. During the first iteration (lines 16–21), the writer should see either the readers' second pattern (which must have been eventually set by one of the preceding readers), or its own second pattern (in case if no readers accessed the buffer since the last buffer flip). In either case, the same pattern is expected to occur during the whole first iteration (line 6), since the writer must have an exclusive access to the structure. (This is guaranteed by waiting for at least one grace period after flipping the buffers.) The second iteration (lines 23–28) writes the second writer's pattern and expects to find the first pattern all the way long.

The writer operates in the reverse order with respect to readers, which increases the probability that racing on the same buffer will be noticed.

Listing 7.2: Writer's buffer access

```
1 static void
2 check_db_writer_verify(buffer_t *buf)
3 {
4         const uint32_t  *const end = buf->b_data - 1;
5         uint32_t        *cur;
6         uint32_t        expected = rcu_access(uint32_t, *buf->b_data);
7 
8         switch (expected) {
        case READER2:   /* FALLTHROUGH */
        case WRITER2:
                break;
        default:
                panic("\tWriter error at 0: Found pattern %x.\n", expected);
        }

        for (cur = buf->b_data + buf->b_size - 1; cur > end; --cur) {
                if (*cur != expected)
                        panic("\tWriter error at %ld: Found pattern %x.\n",
                            (long)(cur - buf->b_data), *cur);
                *cur = WRITER1;
        }
        membar_full();
        for (cur = buf->b_data + buf->b_size - 1; cur > end; --cur) {
                if (*cur != WRITER1)
                        panic("\tWriter error at %ld: Found pattern %x.\n",
                            (long)(cur - buf->b_data), *cur);
                *cur = WRITER2;
        }
}
```

Racing one writer against multiple readers on two buffers should provoke most of the possible race conditions. The RCU API provides no guarantees related to the interaction of writers. This is why a single writer is sufficient to verify that the tested RCU algorithm works as expected.

Since all the threads run under the SDC (System Duty Cycle) scheduling class, all of them can be (and actually are) preempted during their operation, simulating a standard preemptible kernel workload. The SDC class provides basic support for time quanta and a round-robin scheduling behavior. (Normally, preempted kernel threads are pushed to the front of the corresponding per processor per priority queue. The SDC class appends them to the end of the queue instead, which provides the round-robin characteristics.)

All the threads of the testing process print out statistics. For instance, the RCU dual buffer tests print out how many writer iterations have elapsed. There is a constant number of reader threads performing a constant number of reader iterations, whereas the writer runs as long as there are unfinished reader threads. Presumably, the DRCU algorithm achieves the highest number of writer iterations, due to the characteristics of the readers-writer lock, described in subsection 4.1.2. QRCU holds the second place. It is not surprising that the RCU writer is two orders of magnitude slower in this specific scenario. However, it shines in other workloads (much closer to a „real-life“ situation), as shown in the following section 7.2.

Both QRCU and RCU beat DRCU as far as the total duration of the test is concerned, especially with small buffers. This is related to the high overhead caused by readers-writer locks under heavy contention and to their „fairness“ policy.

7.2 Benchmarking the RCU Algorithms

The benchmarking code creates a hash table and then spawns `2 * ncpus` threads that insert, remove and retrieve the hash table items concurrently. All the three operations are performed under an RCU read-side lock. Deleted items are only reclaimed after a grace period has elapsed, so that other concurrent operations do not access inconsistent data.

All the threads access and modify the hash table items in a way similar to what `rcutest` does. Items accessed or removed by the benchmark undergo the read-side or write-side verification procedure, as described in the preceding section 7.1. This also simulates standard data manipulation typical for real workloads.

Each thread running in this benchmark retrieves, inserts and removes hash table items. The ratio between the number of read-only operations (hash table item retrievals) and modifying operations (inserting or removing hash table items) varies between 1:1 and 511:1. The benchmark is run repeatedly with different reader:writer ratios. Each thread first inserts a set of keys from a shared interval (racing with other threads), then inserts another set of keys from a unique interval. Adding the unique keys does not race with anyone when it comes to key conflicts, but there are still races related to the structure of the hash table chains. Following the insertion phase, the thread attempts to remove (possibly unsuccessfully) all the keys from the shared range and then removes its unique keys, reporting an error when any of the latter removals fail. Read-only operations and removals are interleaved. At a ratio of 31:1, for example, 31 read-only operations occur between every two modifying operations.

Each benchmark has two phases. In the first phase, an SMP barrier (contributed by this thesis and described in section 6.2) is used to force all the benchmarking threads to wait for each other after each iteration. (One iteration is the procedure described in the previous paragraph.) This approach is intended to force as many threads as possible to race against each other by performing the same operations simultaneously. The hash table is inspected after each iteration. It must be empty and in a consistent state. In the second phase, all the threads are allowed to perform a certain number of iterations without waiting for each other. Without the barrier after each iteration, threads can race in a less predictable manner closer to real-life situations.

The number of iterations is balanced with respect to the reader:writer ratio, so that the total number of operations (retrievals, insertions and removals) is the same for each run. Since all the operations must compute the hash function, use read-side RCU primitives and traverse a hash chain, this balancing approach is not completely wrong. On the other hand, the modifying operations might be much more time-consuming than retrievals under heavy contention. This implies that it is only meaningful to compare the presented RCU implementations with each other under the same reader:writer ratio. It does not make much sense to compare the results for one RCU implementation and different ratios. Nevertheless, in the context of multiple RCU implementations (and multiple experimental machines), the latter comparison might give a rough estimate of how the reader:writer ratio can influence the performance and scalability of an RCU algorithm.

Payload sizes of the hash table items are chosen pseudo-randomly, varying from 32 bytes to 512 bytes. Short items are much more likely to occur, so that the performance is bound by the RCU primitives rather than by overwriting the payload.

The `rcudemo` module runs the benchmark for each type of RCU and for each of the following five reader:writer ratios: 1:1, 7:1, 32:1, 127:1 and 511:1. On testing machines with more than 8 processors (16 and 24), DRCU caused so much contention that the benchmark had to be tweaked to avoid combining the first three ratios with DRCU. The other two RCU algorithms were benchmarked with all the five ratios on all the three experimental machines.

Total execution time is measured for both phases of each benchmark (the first one with regular rendezvous and the second one with fully parallel execution). However, results are only kept and evaluated for the second („unsynchronized“) phase, since its behavior is expected to be closer to a normal workload. The first phase took a shorter time to complete in most cases, possibly due to the fact that the intensity of races periodically drops near the regular rendezvous. Only the QRCU algorithm (surprisingly) proceeds slightly faster in the absence of rendezvous, especially under the lowest reader:writer ratios.

Admittedly, the QRCU algorithm would perform much better if a separate QRCU instance was created for each hash bucket to protect only one hash chain. However, this would mean that instead of allocating only one pointer per hash bucket, an extra `qrcu_t` structure would have to be allocated for each bucket. It may be interesting to compare such a solution with the global RCU. As this would require changes to both the hash table and the `rcudemo` module, such an experiment is future work. The main purpose of the current benchmark is to test the RCU implementations under heavy contention. Using multiple QRCU instances would reduce the contention, so a performance improvement would not be surprising. (The same experiment could be carried out with DRCU, which could also be instantiated for each hash bucket.)

7.3 Results of the Benchmark

The benchmark has been run on three different multiprocessor machines.

Unfortunately, the lack of time and resources did not make it possible to gather statistically relevant benchmarking data. As the benchmarks take rather long to run and can make the machine almost unusable for tens of minutes to hours (based on processor speed and other factors), it was impossible to perform proper testing in an isolated environment. All the testing machines are shared with many other people and had to stay accessible from the Internet all the time. This limited both the amount of gathered data and the precision of the results.

Furthermore, the benchmark was most frequently run under the `DEBUG` kernel, so that any possible memory consistency and locking issues become obvious as soon as possible. Surprisingly, the performance difference between a `DEBUG` and a `non-DEBUG` kernel was less than 10% on x86_64, but more than 30% on SPARCv9. Obviously, data gathered on the DEBUG kernels can not be reasonably compared side-by-side across multiple platforms due to this fact.

The following tables show the results of two consecutive runs of the benchmark on three different computers running a non-DEBUG kernel, build onnv_145. There are two tables for each machine. The first table lists average values computed from the two runs. The second table contains ratios between corresponding pairs of values. The purpose of the second table is merely to show there seems to be no „striking dissimilarity“ between the two sets of results. As already mentioned, there was not enough data to compute confidence intervals and other statistics.

Types of RCU are denoted as follows:

DRCU The Dummy RCU algorithm using a readers-writer lock instead of a „real“ RCU implementation.

QRCU A variant of the *Sleepable RCU* algorithm ported from Linux. It was initially created by Oleg Nesterov and improved by Paul McKenney.

RCUs The global RCU implementation contributed to the UTS kernel by this thesis. The blocking API is used.

RCUc The global RCU implementation contributed to the UTS kernel by this thesis. The callback API is used.

7.3.1 8 Processors, x86_64

```
$ psrinfo -pv
The physical processor has 4 cores and 8 virtual processors (0-7)
  The core has 2 virtual processors (0 1)
  The core has 2 virtual processors (2 3)
  The core has 2 virtual processors (4 5)
  The core has 2 virtual processors (6 7)
    x86 (GenuineIntel 106E5 family 6 model 30 step 5 clock 1733 MHz)
      Intel(r) Core(tm) i7 CPU       Q 820  @ 1.73GHz
```

Figure 7.1: Testing machine with 8 processors

Configuration of the testing machine can be found in figure 7.1. Results can be seen in table 7.1. DRCU can still be benchmarked on 8 processors, but the other algorithms outperform it in most cases.

Table 7.2 shows the ratios between the first and the second set of measured values. This gives a basic idea of how the consecutive two measurements varied on this particular x86_64 machine with 8 processors.

R:W	Type	Ticks
1:1	DRCU	63233
1:1	QRCU	9653
1:1	RCUs	9687
1:1	RCUc	2839
7:1	DRCU	33888
7:1	QRCU	7192
7:1	RCUs	4353
7:1	RCUc	2365
31:1	DRCU	21322
31:1	QRCU	6225
31:1	RCUs	2971
31:1	RCUc	2239
127:1	DRCU	10064
127:1	QRCU	5990
127:1	RCUs	2624
127:1	RCUc	2183
511:1	DRCU	7117
511:1	QRCU	5945
511:1	RCUs	2271
511:1	RCUc	2081

Table 7.1: RCU benchmark results on an x86_64 machine with 8 processors

R:W	DRCU	QRCU	RCUs	RCUc
1:1	1.0036	1.031	1.005	1.008
7:1	1.0047	1.014	1.006	1.005
31:1	.99789	1.010	1.006	1.009
127:1	.94071	1.004	1.009	1.007
511:1	1.0123	.9988	1.004	.9895

Table 7.2: Ratios of benchmark results from the 8-CPU x86_64 testing machine

```
$ psrinfo -pv
The physical processor has 4 cores and 8 virtual processors (1 3 5 7 9 11 13 15)
  The core has 2 virtual processors (1 9)
  The core has 2 virtual processors (3 11)
  The core has 2 virtual processors (5 13)
  The core has 2 virtual processors (7 15)
    x86 (GenuineIntel 106A5 family 6 model 26 step 5 clock 2533 MHz)
      Intel(r) Xeon(r) CPU           E5540  @ 2.53GHz
The physical processor has 4 cores and 8 virtual processors (0 2 4 6 8 10 12 14)
  The core has 2 virtual processors (0 8)
  The core has 2 virtual processors (2 10)
  The core has 2 virtual processors (4 12)
  The core has 2 virtual processors (6 14)
    x86 (GenuineIntel 106A5 family 6 model 26 step 5 clock 2533 MHz)
      Intel(r) Xeon(r) CPU           E5540  @ 2.53GHz
```

Figure 7.2: Testing machine with 16 processors

7.3.2 16 Processors, x86_64

Configuration of the testing machine can be found in figure 7.2. Results can be seen in table 7.3. With 16 processors, two DRCU benchmarks had to be omitted to conserve time. QRCU seems to be inconvenienced by the increased contention when compared to the 8-CPU machine, but both RCUc and RCUs still perform well.

Table 7.4 shows the ratios between the first and the second set of measured values. This gives a basic idea of how the consecutive two measurements varied on this particular x86_64 machine with 16 processors. Since all the testing machines were exposed to unpredictable network activity, no statistical conclusions can be drawn from these numbers.

7.3.3 24 Processors, SPARCv9

Configuration of the testing machine can be found in figure 7.3. Results can be seen in table 7.5. The 24-thread SPARCv9 machine could not run the first three three of the DRCU benchmarks in a reasonable time. Surprisingly, QRCU became slower with the decreasing amount of concurrent modifications. Both RCUs and RCUc still seem to scale well on 24 SPARCv9 processors.

Table 7.6 shows the ratios between the first and the second set of measured values. This gives a basic idea of how the consecutive two measurements varied on this particular SPARCv9 machine with 24 processors. As already mentioned, the measurement could have been influenced by network activity and other unpredictable events.

R:W	Type	Ticks
1:1	DRCU	—
1:1	QRCU	18677
1:1	RCUs	19275
1:1	RCUc	2900
7:1	DRCU	—
7:1	QRCU	12540
7:1	RCUs	7178
7:1	RCUc	2288
31:1	DRCU	49534
31:1	QRCU	11498
31:1	RCUs	4241
31:1	RCUc	2159
127:1	DRCU	26900
127:1	QRCU	11146
127:1	RCUs	2665
127:1	RCUc	2095
511:1	DRCU	16025
511:1	QRCU	11070
511:1	RCUs	2413
511:1	RCUc	1972

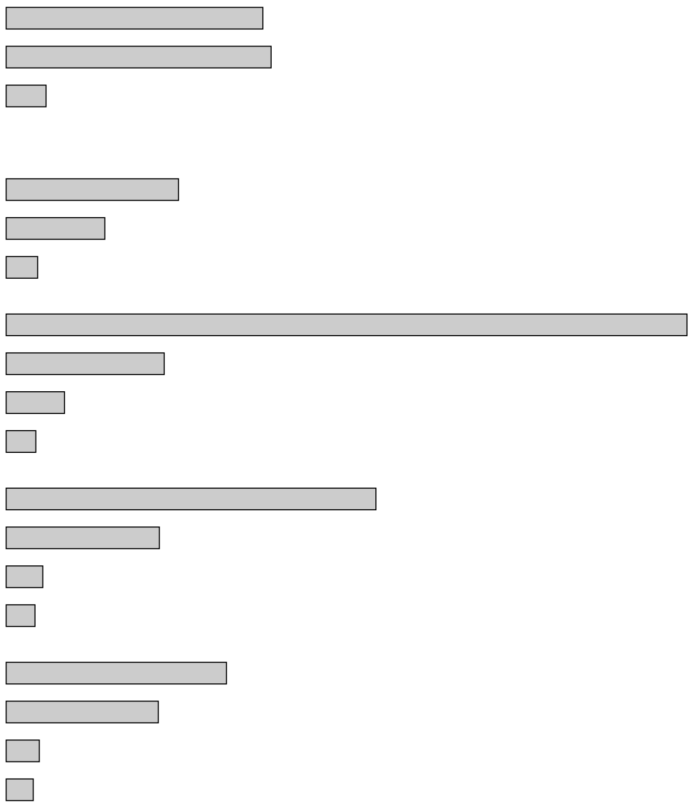

Table 7.3: RCU benchmark results on an x86_64 machine with 16 processors

R:W	DRCU	QRCU	RCUs	RCUc
1:1	—	.95397	1.004	.9952
7:1	—	.89105	.9942	1.006
31:1	1.0173	.87914	.9946	.9821
127:1	1.0000	.88620	.9777	1.018
511:1	.99000	.88965	.9242	1.010

Table 7.4: Ratios of benchmark results from the 16-CPU x86_64 testing machine

R:W	Type	Ticks
1:1	DRCU	—
1:1	QRCU	91235
1:1	RCUs	35918
1:1	RCUc	15649
7:1	DRCU	—
7:1	QRCU	86749
7:1	RCUs	19649
7:1	RCUc	12679
31:1	DRCU	—
31:1	QRCU	98763
31:1	RCUs	14089
31:1	RCUc	12138
127:1	DRCU	56531
127:1	QRCU	105143
127:1	RCUs	13006
127:1	RCUc	11859
511:1	DRCU	40505
511:1	QRCU	108676
511:1	RCUs	12516
511:1	RCUc	11412

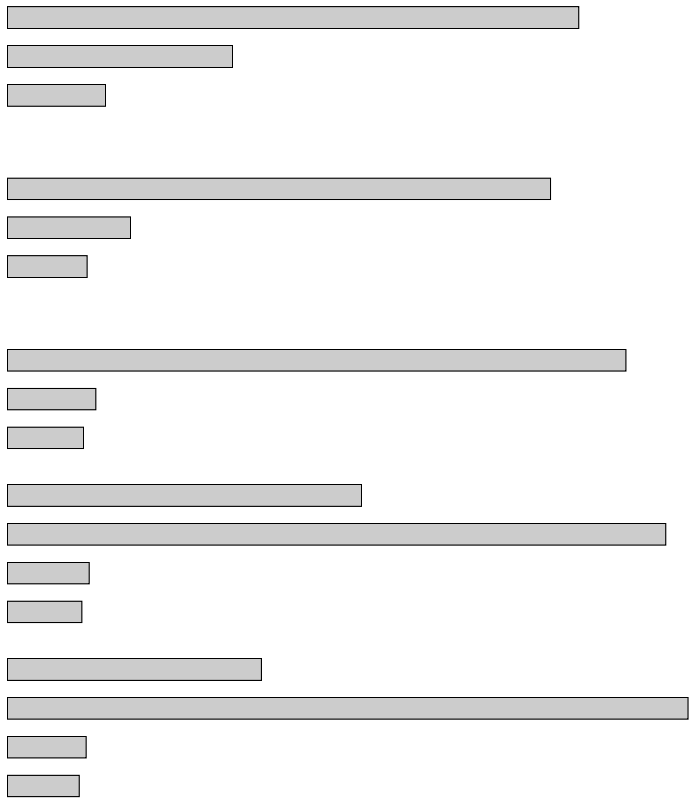

Table 7.5: RCU benchmark results on a SPARCv9 machine with 24 processors

R:W	DRCU	QRCU	RCUs	RCUc
1:1	—	1.0046	.85156	1.0843
7:1	—	1.0080	.86609	1.1154
31:1	—	1.0239	.99398	1.1170
127:1	.99605	1.0585	.99402	1.1084
511:1	1.0056	1.0286	1.0004	1.1173

Table 7.6: Ratios of benchmark results from the 24-CPU SPARCv9 testing machine

```
$ psrinfo -pv
The physical processor has 6 cores and 24 virtual processors (0-23)
  The core has 4 virtual processors (0-3)
  The core has 4 virtual processors (4-7)
  The core has 4 virtual processors (8-11)
  The core has 4 virtual processors (12-15)
  The core has 4 virtual processors (16-19)
  The core has 4 virtual processors (20-23)
    UltraSPARC-T1 (chipid 0, clock 1000 MHz)
```

Figure 7.3: Testing machine with 24 processors

Chapter 8

Future Work

The RCU algorithm implementation is a work in progress. Despite the fact that the RCU algorithm runs and can be benchmarked, there are still issues that need to be dealt with and features that must be added in order to make the algorithm usable in a production environment.

8.1 Read-Side Critical Sections on Offline CPUs

The callback processing (reclaimer) threads are currently common kernel LWPs. These LWPs are CPU-bound, which implies that they must be dynamically spawned and destroyed as CPUs are onlined or offlined. And this is exactly what the current implementation does. This approach would work flawlessly if all the offlined CPUs were also quiesced. Unfortunately, this is not the case in the UTS kernel. It can happen that a CPU being offlined is the last CPU on the system that can process certain I/O interrupts. The offline operation can succeed even in this special case. The offline processor can not run any threads, but it can run interrupt handlers. Interrupt handlers can produce RCU callbacks and contain RCU critical sections.

As already described earlier, the former problem (callbacks produced by interrupt handlers and interrupt threads running on offlined CPUs) has been resolved by choosing an online CPU to which the callbacks produced by the offline one will be relayed. (The offline CPU can not run any threads except interrupt threads, so an RCU reclaimer or detector thread can not run on it.) The chosen online CPU might become offline as well, which forms a singly linked list of offline CPUs with one online CPU at the end. Callbacks produced by all the offline CPUs are relayed to the online CPU at the end of their list.

The latter problem (read-side critical sections in interrupt handlers and interrupt threads running on offline CPUs) has not been resolved yet. The QRCU algorithm can handle this situation due to the absence of centralized callback handling. (In fact QRCU does not provide any callback API.) The RCU algorithm is based on disabling preemption during read-side critical sections. This does not help on offline CPUs, since they are never checked for quiescent states and the detector automatically considers them to be quiesced. (Else the detector would need the possibility to force a context switch on them, which can not happen, since offline CPUs can run no standard threads.)

There are multiple possible solutions to the second problem. Linux uses a complex mechanism that instruments interrupt and NMI handlers so that they announce the activity of the CPU they run on. However, due to the big differences in the scheduler and clock interrupt handling design (described in subsection 5.1.15), this approach can not be taken in UTS.

A naïve solution would introduce a special semaphore for each CPU. The semaphore would be initialized to 1 and it would remain untouched as long as the CPU is online. For offline CPUs, the RCU read-side critical sections would decrement the semaphore (possibly by spinning around `sema_tryp()`) and increment it once they are finished. The detector thread would decrement and immediately re-increment the semaphores of offline CPUs on each grace period boundary. Unfortunately, semaphores are not owned by a thread and do not implement priority inheritance. The high-priority interrupt thread should have the possibility to boost the detector thread's priority in the rare case when it happens to wait for it. This can not be accomplished with semaphores alone.

A better solution would include a transformation of the detector thread and all the CPU-bound reclaimer threads into a form closer to interrupt threads. These threads would be allowed to run on all offline CPUs which are not quiesced, just like interrupt threads. This would solve the problem with read-side critical sections occurring in interrupt handler code.

Interestingly, a race window related to offline (or idle) CPUs executing interrupt handlers with read-side critical sections has existed in the Linux kernel for years. [16] The problem has been resolved with the advent of the hierarchical RCU algorithm. It took years of research to find a solution.

8.2 Removing the Giant Lock

The RCU algorithm should be modified to avoid the centralized global RCU lock used for communication between the detector thread and the reclaimer threads. Although this approach might scale well to tens of CPUs (as long as callback batches take much longer than waiting for the global lock), it would definitely fail on machines with thousands of processors. On these systems, a hierarchical approach would have to be taken. For example, detector threads could be bound to locality groups and announce the detected grace periods in a way similar to the hierarchical RCU algorithm described in subsection 2.1.4. Idle locality groups would not need any RCU-related activity at all, just as idle CPUs in the current implementation.

Since the UTS kernel does not handle scheduling clock ticks on every single CPU, none of the current scalable Linux algorithms can be ported directly. The grace period detection must be based on active detector threads, rather than passive clock tick handler or context switching code instrumentations.

Chapter 9

Conclusion

9.1 Benefits of the Thesis

The accompanying project of this thesis contributed the following:

- A prototype implementation of the RCU mechanism for the UTS kernel, working and fully documented
- A port of the sleepable and preemptible RCU variant called QRCU [9] and a naïve RCU-like algorithm called DRCU for performance comparison
- A non-blocking hash table based around the RCU mechanism, implemented from scratch based on Paul McKenney's description of the algorithm [3]
- Two testing modules that demonstrate the use of RCU and the non-blocking hash table and run performance benchmarks
- Comparison of three variants of RCU running in the UTS kernel on three different multiprocessor machines (x86_64 and SPARCv9)

Additionally, the text of this thesis:

- provides a detailed overview of contemporary RCU algorithms. (chapter 2)
- documents the new implementation of RCU for UTS and compares it with algorithms used in the Linux kernel. (chapter 5)
- describes the non-blocking hash table algorithm (invented by McKenney) and its implementation in UTS. (section 6.1)
- lists a complete set of code snippets using RCU and the non-blocking hash table. (Appendix A)

9.2 Author's Acknowledgements

The author would like to thank Paul McKenney for his Ph. D. thesis and for the numerous papers on RCU and related topics. These papers were the vital source of information necessary to understand all the subtleties of various RCU algorithms. It helped the author realize that there is no „one single correct“ way of thinking about RCU. RCU is not a standard synchronization primitive with a constrained and well-defined set of recommended usage patterns. It is in fact a framework on which various multithreaded algorithm designs can be based.

Another big Thank You goes to the authors of the UTS kernel. The kernel is a sound piece of software with the most advanced observability features in the industry. Despite the complexity of contemporary hardware and state-of-the-art kernel algorithms, the source code is easy to read and understand. It is therefore surprising that relatively few academic projects are based on the UTS kernel.

Appendix A

Using RCU

This chapter lists some elementary code snippets that clarify the relationship between RCU and the protected data structure. They perform operations on a non-blocking hash table protected by RCU. Only QRCU and the global RCU implementation are considered here. DRCU is not useful for anything but benchmarking.

A.1 Common Data Structures

All the examples are based on the data structure definitions shown in figure A.1. For simplicity, the `container_of()` macro known from the Linux kernel is used in the following examples. (No such macro is defined in UTS.) This macro casts a pointer to a structure member (the first argument) to a pointer to the containing structure. The type of the containing structure is specified by the second argument. The third argument contains the name of the member the first argument points at.

```
typedef struct scompound {
        ht_t            s_ht;           /* hash table binding */
        payload_t       s_payload;      /* user-defined payload data */
} scompound_t;

typedef struct ccompound {
        ht_t            c_ht;           /* hash table binding */
        rcu_t           c_rcu;          /* RCU callback binding */
        payload_t       c_payload;      /* user-defined payload data */
} ccompound_t;
```

Figure A.1: Data structures used in the examples

The `ht_t` and `rcu_t` data structures are part of the non-blocking hash table and RCU API, respectively. The `payload_t` type identifier stands for an arbitrary user-defined data type.

```
int read(ht_map_t *map, ht_key_t key, payload_t *result);
int insert(ht_map_t *map, ht_key_t key,
    const payload_t *source, payload_t *result);
int remove(ht_map_t *map, ht_key_t key, payload_t *result);
```

Figure A.2: Transparent mapping operations

A.2 Operation Definitions

The hash map operations shown in this section are listed in figure A.2.

The `read()` operation looks up `key` in a hash table `map`. A nonzero value is returned when `key` is found, zero otherwise. When a matching key is found and `result` is not null, the data mapped to `key` is copied into the area referenced by `result`.

The `insert()` operation attempts to establish a mapping of `key` to the data referenced by `source` in the hash table `map`. A nonzero value is returned when `key` is not found (and the operation succeeds), zero otherwise. When a conflicting key is found and `result` is not null, the data mapped to the conflicting key is copied into the area referenced by `result`.

The `remove()` operation removes the mapping of `key` from the hash table `map`. A nonzero value is returned when `key` is found (and the removal succeeds), zero otherwise. When a matching key is found and `result` is not null, the data mapped to `key` is copied into the area referenced by `result`.

A.3 Hash Table Operations with RCU

Listings A.1, A.2, A.3 and A.4 show the three basic operations on a hash table with the assistance of RCU. Note that all the manipulation with structures accessible from the hash table can only happen under protection of the read-side RCU primitives.

There are two versions of the `remove()` operation. The first one uses the blocking API and the second one enqueues a callback and proceeds immediately. Only one version of the other two operations is shown. In a real implementation, references to `scompound_t` in the `read()` and `insert()` functions would have to be replaced with `ccompound_t` to match the type of hash table items. The rest of the source code would be identical.

The `insert()` and `remove()` functions could be optimized to avoid copying the data. In that case, the caller would have to be aware of how the data records are stored.

Listing A.1: The `read()` operation with global RCU

```
int
read(ht_map_t *map, ht_key_t key, payload_t *result)
{
        int             index;
        ht_t            *retval;
        payload_t       *data;

        rcu_read_lock();        // <<< LOCK
        retval = ht_get(map, key);
        if (!retval) {
                qrcu_read_unlock(qrcu, index);
                return (0);
        }
        if (result) {
                data = &container_of(retval, scompound_t, s_ht)->s_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        rcu_read_unlock();      // <<< UNLOCK

        return (1);
}
```

Listing A.2: The `insert()` operation with global RCU

```
int
insert(ht_map_t *map, ht_key_t key, const payload_t *source, payload_t *result)
{
        int             index;
        ht_t            *retval;
        scompound_t     *record;
        payload_t       *data;

        record = kmem_alloc(sizeof(scompound_t), KM_SLEEP);
        bcopy(source, &record->s_payload, sizeof(payload_t));

        rcu_read_lock();        // <<< LOCK
        retval = ht_add(map, key, &record->s_ht);
        if (retval && result) {
                data = &container_of(retval, scompound_t, s_ht)->s_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        rcu_read_unlock();      // <<< UNLOCK

        if (retval) {
                kmem_free(record, sizeof(scompound_t));
                return (0);
        }
        return (1);
}
```

Listing A.3: The `remove()` operation with global RCU (using the blocking API)

```
int
remove(ht_map_t *map, ht_key_t key, payload_t *result)
{
        int             index;
        ht_t            *retval;
        scompound_t     *record;
        payload_t       *data;

        rcu_read_lock();        // <<< LOCK
        retval = ht_del(map, key);
        rcu_read_unlock();      // <<< UNLOCK

        if (!retval) {
                return (0);
        }
        record = container_of(retval, scompound_t, s_ht);
        rcu_synchronize();      // <<< SYNCHRONIZE
        if (result) {
                data = &record->s_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        kmem_free(record, sizeof(scompound_t));
        return (1);
}
```

Listing A.4: The `remove()` operation with global RCU (using the callback API)

```
void
reclaim(rcu_t *rcu) {
        ccompound_t     record;

        record = container_of(rcu, ccompound_t, c_rcu);
        kmem_free(record, sizeof(ccompound_t));
}

int
remove(ht_map_t *map, ht_key_t key, payload_t *result)
{
        int             index;
        ht_t            *retval;
        ccompound_t     *record;
        payload_t       *data;

        rcu_read_lock();        // <<< LOCK
        retval = ht_del(map, key);
        rcu_read_unlock();      // <<< UNLOCK

        if (!retval) {
                return (0);
        }
        record = container_of(retval, ccompound_t, c_ht);
        if (result) {
                data = &record->c_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        rcu_call(               // <<< CALLBACK
            reclaim, &record->c_rcu,
            RCU_WEIGHT(1, sizeof(ccompound_t))
        );
        return (1);
}
```

A.4 Hash Table Operations with QRCU

Listings A.5, A.6 and A.7 show the three basic operations on a hash table with the assistance of QRCU. Note that all the manipulation with structures accessible from the hash table can only happen under protection of the read-side lock.

The `insert()` and `remove()` functions could be optimized to avoid copying the data. The caller would have to be aware of how the data records are stored.

Listing A.5: The `read()` operation with QRCU

```
int
read(qrcu_t *qrcu, ht_map_t *map, ht_key_t key, payload_t *result)
{
        int             index;
        ht_t            *retval;
        payload_t       *data;

        index = qrcu_read_lock(qrcu);   // <<< LOCK
        retval = ht_get(map, key);
        if (!retval) {
                qrcu_read_unlock(qrcu, index);
                return (0);
        }
        if (result) {
                data = &container_of(retval, scompound_t, s_ht)->s_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        qrcu_read_unlock(qrcu, index);  // <<< UNLOCK

        return (1);
}
```

Listing A.6: The `insert()` operation with QRCU

```
int
insert(qrcu_t *qrcu, ht_map_t *map, ht_key_t key,
    const payload_t *source, payload_t *result)
{
        int             index;
        ht_t            *retval;
        scompound_t     *record;

        record = kmem_alloc(sizeof(scompound_t), KM_SLEEP);
        bcopy(source, &record->s_payload, sizeof(payload_t));

        index = qrcu_read_lock(qrcu);   // <<< LOCK
        retval = ht_add(map, key, &record->s_ht);
        if (retval && result) {
                data = &container_of(retval, scompound_t, s_ht)->s_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        qrcu_read_unlock(qrcu, index);  // <<< UNLOCK

        if (retval) {
                kmem_free(record, sizeof(scompound_t));
                return (0);
        }
        return (1);
}
```

Listing A.7: The `remove()` operation with QRCU

```
int
remove(qrcu_t *qrcu, ht_map_t *map, ht_key_t key, payload_t *result)
{
        int             index;
        ht_t            *retval;
        scompound_t     *record;
        payload_t       *data;

        index = qrcu_read_lock(qrcu);   // <<< LOCK
        retval = ht_del(map, key);
        qrcu_read_unlock(qrcu, index);  // <<< UNLOCK

        if (!retval) {
                return (0);
        }
        record = container_of(retval, scompound_t, s_ht);
        qrcu_synchronize(qrcu);         // <<< SYNCHRONIZE
        if (result) {
                data = &record->s_payload;
                bcopy(data, result, sizeof(payload_t));
        }
        kmem_free(record, sizeof(scompound_t));
        return (1);
}
```

Appendix B

Building OS/Net with RCU

This chapter is a brief manual that provides instructions on building the modified UTS kernel from source and installing it. Readers are expected to be familiar with the OS/Net source tree and the build process.

B.1 Patching the OS/Net Sources

The OS/Net source code can be downloaded from a Mercurial repository and the RCU patch can be applied using the `hg import` command. Preferably, a tagged build should be used rather than the tip revision. Figure B.1 shows how the patch can be applied to build 145. The patch can be found on the DVD supplied with this thesis.

```
$ mkdir onws && cd onws
$ hg clone ssh://anon@hg.opensolaris.org/hg/onnv/onnv-gate osnet145
$ cd osnet145
$ hg update -r onnv_145
$ hg import - < /path/to/rcu.patch
```

Figure B.1: Patching the OS/Net source code

B.2 Building the Modified Kernel

The OS/Net build process is described in [21]. It is always recommended to build the full OS/Net to make sure all the userspace tools are fully compatible with the modified UTS kernel.

When the modified OS/Net is successfully installed using the `onu` script and booted, a message saying `RCU: detector starting` will appear on the console during the boot process.

At the time of this writing, the OS/Net build process still requires two packages of closed-source binaries. [21, 22] It is advisable to download the version of binaries that matches the chosen OS/Net build. Closed-source binaries are released for each build. Nightly snapshots matching the repository tip revision are also available for download. Furthermore, it is necessary to install numerous other software packages and compile the latest packaging bits. [21]

The DVD supplied with this thesis contains helper scripts that can be used to obtain a usable build environment from a fresh source code checkout. Directories with the OS/Net source trees cloned from Mercurial are expected to be located in the same directory as the scripts. Figure B.2 is based on this assumption.

The `updateclosed` script in figure B.2 downloads the latest nightly build of closed binaries (both `DEBUG` and `non-DEBUG` versions) and extracts them into the required subdirectory (`osnet`) containing an OS/Net source tree. It also updates the source tree to the latest revision (`hg pull; hg update`).

When using revisions other than the tip revision (such as a tagged build, `onnv_145` for instance), it is necessary to download and install the matching binary files manually and the `updateclosed` command must **not** be run.

The `bootstrap` script compiles and installs the latest build tools and scripts that run the whole build process. The `opensolaris.sh` file is called an *environment file* and specifies your local paths and other settings. The Developer's Reference [21] describes what this file has to contain and where to obtain a usable template.

```
$ cd onws
$ ./updateclosed osnet
$ ./bootstrap osnet osnet/opensolaris.sh
```

Figure B.2: The „bootstrap“ procedure

Both scripts mentioned in figure B.2 have been created by the author of this thesis. They are provided for reference only. They might be bound to the author's local workspace and unusable on other machines. They can be either used as human-readable task lists or customized to match other users' directory structure and working environment.

After installing the official OS/Net Build Environment, installing all the necessary packages (especially `developer/opensolaris/osnet`), checking out and patching the source code, installing the closed-source binaries, creating an environment file and „bootstrapping“ the build tools, the modified OS/Net can be compiled.

Details about the build process, a complete list of prerequisites and links to other sources of information can be found in the `README` file on the DVD supplied with this thesis.

The DVD also contains **pre-built binary packages** for both SPARC and x86. These packages are stored in the form of an IPS repository directory. They can be installed using the `onu` script.

Bibliography

[1] McKenney P., Appavoo J., Kleen A., Krieger O., Russell R., Sarma D., Soni M.: *Read-Copy Update*; Ottawa Linux Symposium; `http://lse.sourceforge.net/locking/rcu/rclock_OLS.2001.05.01c.sc.pdf`; 2001

[2] Soules C., Appavoo J., Hui K., Da Silva D., Ganger G., Krieger O., Stumm M., Wisniewski R., Auslander M., Ostrowski M., Rosenburg B., Xenidis J.: *System Support for Online Reconfiguration*; Carnegie Mellon University, University of Toronto, IBM T. J. Watson Research Center; 2003

[3] McKenney P.: *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*; Ph. D. Thesis; Oregon State University; 2004

[4] McKenney P.: *Kernel Korner - Using RCU in the Linux 2.5 Kernel*; Linux Journal; `http://www.linuxjournal.com/article/6993`; 2003

[5] McKenney P.: *Memory Ordering in Modern Microprocessors, Part I*; Linux Journal; `http://www.linuxjournal.com/article/8211`; 2005

[6] McKenney P.: *Memory Ordering in Modern Microprocessors, Part II*; Linux Journal; `http://www.linuxjournal.com/article/8212`; 2005

[7] McDougall R., Mauro J.: *Solaris Internals*; Prentice Hall; Westford; 2006

[8] McDougall R., Mauro J., Gregg B.: *Solaris Performance and Tools*; Prentice Hall; Westford; 2006

[9] McKenney P.: *Using Promela and Spin to verify parallel algorithms*; Linux Weekly News; `http://lwn.net/Articles/243851/`; 2007

[10] McKenney P.: *The design of preemptible read-copy-update*; Linux Weekly News; `http://lwn.net/Articles/253651/`; 2007

[11] McKenney P.: *What is RCU, Fundamentally?*; Linux Weekly News; `http://lwn.net/Articles/262464/`; 2007

[12] McKenney P.: *What is RCU? Part 2: Usage*; Linux Weekly News; `http://lwn.net/Articles/263130/`; 2007

[13] Faulkner R.: *double checked locking code needs a memory barrier*; Bug Report; `http://bugs.opensolaris.org/bugdatabase/view_bug.do?bug_id=6513516`; 2007

[14] McKenney P.: *RCU part 3: the RCU API*; Linux Weekly News; `http://lwn.net/Articles/264090/`; 2008

[15] McKenney P.: *Sleepable Read-Copy Update*; Linux Technology Center, IBM Beaverton; `http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf`; 2008

[16] McKenney P.: *Hierarchical RCU*; Linux Weekly News; `http://lwn.net/Articles/305782/`; 2008

[17] Marejka R.: *Atomic SPARC: Using the SPARC Atomic Instructions*; Sun Developer Network; `http://developers.sun.com/solaris/articles/atomic_sparc/`; 2008

[18] McKenney P.: *RCU: The Bloatwatch Edition*; Linux Weekly News; `http://lwn.net/Articles/323929/`; 2009

[19] Desnoyers M.: *Low-Impact Operating System Tracing*; Ph. D. Thesis; École Polytechnique de Montréal; 2009

[20] Solter N., Jelinek G., Miner D.: *OpenSolaris Bible*; Wiley Publishing; Indianapolis; 2009

[21] Various Authors: *OpenSolaris ON Developer's Reference Guide*; OpenSolaris WIKI; `http://hub.opensolaris.org/bin/view/Community+Group+on/devref_toc`; 2010

[22] Davis E.: *How To Build OpenSolaris*; Blog Post; `http://insanum.com/blog/2010/06/08/how-to-build-opensolaris`; 2010

[23] Wikipedia: *Memory Ordering*; `http://en.wikipedia.org/wiki/Memory_ordering`