1-2005

RCU Semantics: A First Attempt

Paul E. McKenney

Jonathan Walpole
Portland State University

RCU Semantics: A First Attempt

Paul E. McKenney
IBM Linux Technology Center
15400 SW Koll Parkway
Beaverton, OR 97006
Paul.McKenney@us.ibm.com
http://www.rdrop.com/users/paulmck

Jonathan Walpole
Computer Science Department
Portland State University
walpole@cs.pdx.edu

Draft of 2005/01/30 16:00

Abstract

There is not yet a formal statement of RCU (read-copy update) semantics. While this lack has thus far not been an impediment to adoption and use of RCU, it is quite possible that formal semantics would point the way towards tools that automatically validate uses of RCU or that permit RCU algorithms to be automatically generated by a parallel compiler. This paper is a first attempt to supply a formal definition of RCU. Or at least a semi-formal definition: although RCU does not yet wear a tux (though it does run in Linux), at least it might yet wear *some* clothes.

1 Introduction

RCU has has been applied to at least five operating systems, three of which have seen extensive production use (VM/XA, DYNIX/ptx, and Linux) [3, 9, 13], and the other two being prominent research operating systems (K42 and Tornado) [1]. RCU was independently invented at least six times [7, 3, 6, 12, 9, 1], and, though there has been some academic work on RCU, the bulk of the work has been by practitioner for practitioners.

Although RCU has made great strides, the lack of academic focus has left RCU bereft of the sound theoretical basis required for analysis and tooling. In addition, use of RCU has been undisciplined. This lack of discipline has left a legacy of innovative applications of RCU, which has the benefit of having explored a large expanse of the RCU landscape, but this variety also makes theoretical analysis more complex.

To make the analysis more tractable and useful, this paper therefore imposes some assumptions. These assumptions are chosen carefully so that most uses of RCU are covered by the resuting analysis, and so that this analysis can be readily extended.

2 Concepts

RCU is used by "threads" operating on "data elements". A thread might be a CPU, process, task, coroutine, interrupt handler, or any other abstraction that represents a locus of control and corresponding local state.

Data elements correspond to C structs, form a set E of elements, each of which can be modeled as a tuple with "fields", "pointers", and "locks". Fields can be any sort of value, such as integer, floating point, or string, while pointers are references to other data elements. A subset G of these data element are *global*, they may be referenced by any thread at any time. Non-global data elements must be referenced indirectly, starting with a pointer of a data element that is a member of G. The sequence of data elements that was traversed to reach a given data element is called a "reference path".

Locks serialize updates to corresponding fields. The mapping from a given lock to the fields that it guards is peculiar to the algorithm referencing these fields; this mapping can be thought of as a key part of the locking design for these fields.

Threads are not permitted to reference data elements in an undisciplined manner, instead, threads can only reference data elements while in a "critical section". Critical sections are delimited by lock acquisitions or by the boundaries of RCU read-side critical sections. In the Linux kernel, the `spin_lock()` and `spin_unlock()` acquire and release locks, while `rcu_read_lock()` and `rcu_read_unlock()` demark the beginning and end of RCU read-side critical sections.

3 Definitions

The value of a data element can vary with time, and is denoted by the function $V(e, f, t)$ for field f of element e at time t. By convention, $V(e, l, t)$ represents a reference to one of element e's locks, and, similarly, $V(e, p, t)$ represents a reference to one of element e's pointers. Note that pointers can either reference a given data element e, or can be "NULL", not referencing any data element.

Each thread enters and exits RCU read-side critical sections, and these critical sections can be nested. The number of times that a thread T has entered an RCU read-side critical section up through time t is given by $R_a(T, t)$, and the number of times that this same thread has exited an RCU read-side critical section over that same time period is given by $R_r(T, t)$. Therefore, the expression:

$$R_a(T, t) - R_r(T, t) \tag{1}$$

will be non-zero iff thread T is in an RCU read-side critical section at time t.

An RCU read-side critical section may then be defined as a time interval $[t_a, t_b]$ such that:

$$R_a(T, t) > R_r(T, t) \quad \forall t : t_a \leq t \leq t_b \tag{2}$$

Multiple threads may be in RCU read-side critical sections concurrently.

Lock-based critical sections may be used to serialize updates to RCU-protected variables. These are represented in a manner similar to that for RCU read-side critical sections: the number of times that a given thread T has acquired and released a given lock l up to time t is given by $L_a(T, l, t)$ and $L_r(T, l, t)$, respectively. A lock-based critical section is then a time interval $[t_a, t_b]$ such that:

$$L_a(T, l, t) > L_r(T, l, t) \quad \forall t : t_a \leq t \leq t_b \tag{3}$$

Only one thread may hold a given lock at a given time, so that:

$$L_a(T_1, l, t) > L_r(T_1, l, t) \rightarrow L_a(T, l, t) = L_r(T, l, t) \quad \forall T \neq T_1 \tag{4}$$

As noted earlier, any data element $e \notin G$ can be referenced only via a reference path, which can be represented as a series of n data elements:

$$e_1, e_2, e_3, ..., e_n \tag{5}$$

where $e_1 \in G$ and $e_{i+1} = V(e_i, p, t_i)$ and $t_{i+1} > t_i$. Normally, such a reference path must be traversed within a critical section $[t_a, t_b]$, in other words, $t_a \leq t_1 \leq t_n \leq t_b$. It is normally not permitted to maintain a reference to any data element $e \notin G$ except when within a corresponding critical section.

If a reference path is traversed within a lock-based critical section $[t_a, t_b]$, we know that:

$$V(e, p, t) = V(e, p, t_a) \quad \forall t_a \leq t \leq t_b \tag{6}$$

Therefore, reference paths are stable within a lock-based critical section.

However, RCU read-side critical sections do not exclude updates. It is quite possible that a data element e_i in a given reference path is no longer reachable at some time $t \geq t_i$, in other

words, there is no reference path extending from a member of G to e_i. However, the use of grace periods guarantees that, within a given RCU read-side critical section $[t_a, t_b]$, if data element e was reachable at time $t_a \leq t_1 < t_b$, but was not reachable at some time $t_1 < t_2 < t_b$, then it cannot be reachable at any time $t_2 < t \leq t_b$.

It is therefore possible that a thread T within an RCU read-side critical section $[t_a, t_b]$ might reference a data element e_i via some reference path $e_1, e_2, e_3, ..., e_n$ that is no longer reachable. This element e_i is said to represent "stale data". In addition, some other thread T' might concurrently find that element e_1 instead references some other element e_2'. This discrepancy between element e_2 and e_2' is said to represent "inconsistent data". Many useful algorithms tolerate both stale and inconsistent data, while many other algorithms may be transformed to tolerate stale and inconsistent data [8].

4 Simplifying Assumptions

In the most common use of RCU, elements are either added to or removed from a linked list. The non-pointer fields within each such element are not modified while that element is on the list, and are in fact left unchanged for a full grace period after removal of that element from the list. This means that during an RCU read-side critical section $[t_a, t_b]$, for any field f of any RCU-protected element e:

$$V(e, f, t) = V(e, f, t_a) \quad \forall t : t_a \leq t \leq t_b \tag{7}$$

This assumption may be relaxed, but such relaxation is outside the scope of this paper.

5 Grace Period

Equation 7 requires that the fields of an RCU-protected data element remain constant as long as they might be referenced by a thread executing an ongoing RCU read-side critical section. One way to achieve this would be to preserve any element rendered inaccessible, but limited memory renders this approach impractical. Therefore, this requirement imposes the need for a grace period, either in the style of RCU or of hazard pointers [10, 11, 4, 5], that determines when a given data element may safely be recycled. The following sections derive expressions for each of these possibilities.

5.1 RCU Grace Period

Once a given data element has been rendered inaccessible, it may be recycled as soon as all RCU read-side critical sections have completed. If the data element was removed at time t_r, it may be recycled at time t_g, which is implicitly defined by the following:

$$\forall T \exists t_q : R_a(T, t_q) = R_r(T, t_q) \bigwedge t_r \leq t_q \leq t_g \tag{8}$$

This equation constrains each thread T to exit all RCU read-side grace periods before time t_g. The per-thread time t_q corresponds to the RCU concept of "quiescent state", which is a state in which a given thread can be guaranteed not to be accessing any RCU-protected data structure.

5.2 Hazard-Pointer Grace Period

Hazard pointers are used to record all data structures to which a given thread holds reference. When a given data element has been rendered inaccessible, as soon as that data element is no longer referenced by any hazard pointers, it may be recycled. Assigning a given data element to a hazard pointer explicitly acquires a reference to that data element, and later overwriting that same hazard pointer releases the reference.

In contrast to RCU, in which read-side critical sections are specific to threads, the hazard-pointer equivalent is specific to the combination of a thread and a data element. The count of the number of times that a thread T has assigned a hazard pointer to reference a data element

e up through time t is given by $H_a(T,e,t)$ and the number of times that such a value has been overwritten is given by $H_r(T,e,t)$. When

$$H_a(T,e,t) = H_r(T,e,t) \tag{9}$$

The thread T is not referencing element e, and can therefore tolerate its being recycled.

By analogy with Equation 8, a data element e that has been rendered inaccessible at time t_r may safely be recycled at time t_g, which is implicitly defined by:

$$\forall T \exists t_q : H_a(T,e,t_q) = H_r(T,e,t_q) \bigwedge t_r \leq t_q \leq t_g \tag{10}$$

5.3 Multiple RCU Grace Periods

The Linux 2.6 kernel has two instances of RCU, each with its own set of quiescent states. One of these RCU instances is designed for use in process-level algorithms, and the other for algorithms that run with interrupts disabled.

One can generalize this to a set of RCU instances $\{R_1, R_2, ...R_n\}$, where each such instance R_i has an associated set of quiescent states q_i and a set of data elements d_i that are protected by that instance of RCU.

Once a given data element protected by instance i of RCU has been rendered inaccessible, it may be recycled as soon as all of the i^{th} instance's RCU read-side critical sections have completed. If the data element was removed at time t_r, it may be recycled at time t_g, which is implicitly defined by the following:

$$\forall T \exists t_q : R_{i,a}(T,t_q) = R_{i,r}(T,t_q) \bigwedge t_r \leq t_q \leq t_g \tag{11}$$

Here $R_{i,a}(T,t)$ counts the number of RCU read-side critical-section entries by thread T up to time t for RCU instance i, and $R_{i,r}(T,t)$ similarly counts the number of RCU read-side critical section exits. Again, the per-thread times t_q corresponds to the per-thread quiescent states, but for instance i of RCU.

6 Modeling of RCU Operations

Common use of RCU makes use of the following operations:

1. Insertions of a fully initialized element into a linked list.
2. Deletion by first removing an element from a linked list, then disposing of it following a grace period.
3. Updating a linked list by making a copy of a existing element, then atomically replacing the original with the updated version.

The following sections describe how each of these operations is modeled.

6.1 Insertion

In the case of insertion, a new element e (which must not be referenced by any thread) is allocated, each field f and pointer p is initialized, a memory barrier is executed on architectures not featuring sequential consistency, and then the element is inserted into the list. This process is illustrated by Figure 1.

While element e is being initialized, its fields f are subject to change, but, during this time, the element remains inaccessible, so that no thread in an RCU read-side critical section is able to observe these changes. Once the element is inserted into the list, it becomes accessible to such threads, but by this time its fields are constant, as required.

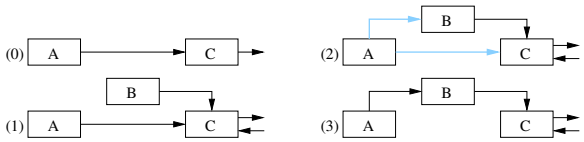

Figure 1: Atomic Insertion Into a Linked List

6.2 Deletion

In the case of deletion, an element e is first removed from its list at time t_r, rendering it inaccessible to all future RCU read-side critical sections $[t_a, t_b], t_a > t_r$. Then Equation 8 may be used to determine a time t_g after which element e may be safely recycled (which can result in all its fields being modified). After time t_g, all RCU read-side critical sections that started on or before time t_r are guaranteed to have ended. This process is illustrated by Figure 2.

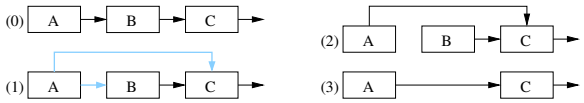

Figure 2: Atomic Deletion From a Linked List

Note that any thread that maintains a reference to element e after time t_r will be processing stale data.

6.3 Update

This section first considers the special case where only a single thread is permitted to do updates, extends this result to cover lock-mediated updates, then finally examines update mediated by a simple form of non-blocking synchronization (NBS). More elaborate forms of NBS are outside of the scope of this paper.

6.3.1 Update From Single Thread

In the case of update, a new element e_n is allocated and initialized by copy from the old version e_o. The new version is then updated as desired, then inserted into the list in place of the old version. This insertion is accomplished by updating any pointer p that references e_o to instead reference e_n. In other words, if this replacement occurs at time t_r:

$$\forall e \forall p : V(e, p, t_r - \epsilon) = e_o \rightarrow V(e, p, t_r) = e_n \tag{12}$$

where ϵ is the smallest unit of time, roughly corresponding to the clock period of physical computer systems. Once this replacement has been accomplished, the old element e_o may be recycled at a later time t_g when all RCU read-side critical sections starting before time t_r have completed, again determined using Equation 8.

Note that at time t_r there are two versions of the same data element, namely e_o and e_n. If thread T_o entered its RCU read-side critical section just before t_r and another thread T_n did so just after t_r, then it is quite possible that T_o will be referencing e_o concurrently with T_n referencing e_n. This provides an example of inconsistent references to multiple versions of a given data element.

6.3.2 Locking-Mediated Update

Locking-mediated updates make use of a lock l that is a member of some distinguished data element e. There is an associated locking design that may be represented by a set S_l containing fields, pointers, and locks associated with any number of data elements, not necessarily including

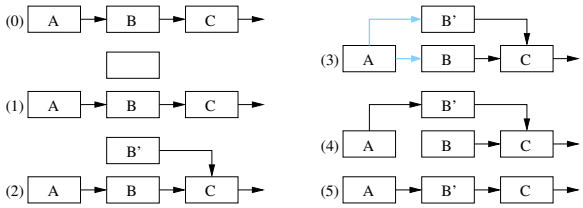

Figure 3: Atomic Update of an Element of Linked List

element e. By convention, a thread T is permitted to modify the members of S_l at time t only if T holds lock l at time t, in other words, iff:

$$L_a(T, l, t) > L_r(T, l, t) \tag{13}$$

By Equation 4, at time t, only threat T may hold the lock, and therefore no other thread will be concurrently modifying the members of S_l.

Since only a single thread is permitted to update the data structure at a time, the remainder of the analysis proceeds as outlined in Section 6.3.1.

6.3.3 NBS-Mediated Update

A wide variety of NBS update algorithms have been proposed, so it is beyond the scope of this paper to analyze all of them. For simplicity, this section will focus on a simple case where a linked data structure is updated via full replacement, the so-called "small-object" approach.. Furthermore, this section will assume that readers and updaters make use of either RCU or hazard pointers in order to avoid the ABA problem [14], in which an element that has been freed and subsequently reused is erroneously used as if it had retained its old identity. Both RCU and hazard pointers avoid the ABA problem by guaranteeing that no structure will be reused until all threads have dropped any references that they might have to it. It also permits any external pointer p to the data structure to be used as a version number for that data structure [2].

Updates to a given data structure referenced by an external pointer p use the following procedure:

1. Copy the value of p to a local variable v_1.
2. Allocate data elements.
3. Copy the data structure referenced by p to the newly allocated data elements, with the copy referenced by a local variable v_2.
4. Update the newly created copy.
5. Atomically compare p to v_1 and set p's value to v_2 if equal (indicating successful update).
6. If the comparison in the previous step failed (was not equal), free up the data elements referenced by v_2 and retry.

Given a set S_T of threads that are concurrently attempting to update the data structure, only one of them can succeed in step 5. The remainder will need to retry the update.

This means that only one thread at a time may update the data structure, so the remainder of the analysis proceeds as described in Section 6.3.1.

This analysis may be extended to cover a wider variety of NBS algorithms, if desired.

7 Conclusions

This paper has provided an abstraction that captures the semantics of a common use of RCU. This semantic analytically demonstrates the well-known presence of stale and inconsistent data in RCU-based algorithms, and also demonstrates the mathematical necessity of quiescent states,

which were heretofore believed to be a artifact of RCU implementations. In addition, a deep analogy between RCU and hazard pointers was uncovered.

More work is needed to extend this model of RCU semantics to cover more specialized uses of RCU.

8 Acknowledgments

This document would not be complete if it did not recognize the debt owed to Robert Bauer and Bart Massey, who managed to convince one of the authors (Paul) that some good might in fact come from deriving the formal semantics of RCU.

References

[1] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100.

[2] GREENWALD, M., AND CHERITON, D. R. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 123–136.

[3] HENNESSY, J. P., OSISEK, D. L., AND SEIGH II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168 (lapsed), US Patent and Trademark Office, Washington, DC, February 1989.

[4] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing* (October 2002), pp. 339–353.

[5] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)* (July 2003), pp. 92–101.

[6] JACOBSON, V. Avoid read-side locking via delayed free. Verbal discussion, September 1993.

[7] KUNG, H. T., AND LEHMAN, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382.

[8] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[9] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

[10] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architecture* (August 2002), pp. 73–82.

[11] MICHAEL, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing* (August 2002), pp. 21–30.

[12] PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. Optimistic incremental specialization: Streamlining a commercial operating system. In *15th ACM Symposium on Operating Systems Principles (SOSP'95)* (Copper Mountain, CO, December 1995), pp. 314–321.

[13] TORVALDS, L. Linux 2.5.43. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2` [Viewed June 23, 2004], October 2002.

[14] TREIBER, R. K. Systems programming: Coping with parallelism. RJ 5118, April 1986.