Applications of RCU

Christian Bewermeyer
Friedrich-Alexander-Universität Erlangen-Nürnberg
christian.bewermeyer@fau.de

ABSTRACT

The RCU synchronization mechanism has been the subject of a fair amount of research over the past few years. This paper takes a look at RCU from a application-oriented perspective and surveys the usage of RCU in real-world systems. The first part of this work deals with the Linux RCU API while the second part discusses user-level implementations. In addition, this paper contrasts the advantages and drawbacks of using RCU, helping system developers in the choice of the proper synchronization primitives.

1. A SHORT INTRODUCTION TO RCU

The purpose of this section is to familiarize the reader with the theoretic foundations of RCU which are required to understand the rest of this paper. After a brief overview of the main properties of RCU, we will introduce the concept of grace periods which is of crucial importance to RCU. The sections ends with a short example.

RCU (read-copy-update) is a synchronization mechanism that allows readers to unconditionally make forward progress, even in the presence of updaters. Because of this property, RCU differs greatly from conventional mutual exclusion synchronization techniques such as read-write locks in which write operations cause a blockade of all readers [3]. RCU thus allows concurrency between multiple readers and a single updater (multiple updaters must be serialized with other mechanisms such as spinlocks).

Conceptually, the fundamental idea of RCU is to maintain multiple versions of objects and ensure that each version remains intact as long as there are readers that hold references to it and use the data contained in that version of the object [10]. RCU also provides efficient mechanisms to publish new versions of an object and for detecting when old versions can safely be freed. The main objective of RCU is to supply low-overhead read-side primitives in order to make read operations as fast as possible. In addition, readers must not adhere to complicated locking protocols and can avoid precautions such as disabling interrupts.

A RCU implementation must ensure that a version of a data structure is not destroyed as long as there exist threads with read access to it. Of course, the question that now arises is how RCU can determine that no more readers are accessing a given version of a data object in order to free the associated memory. The answer is best described using a graphical representation (see Figure 1). In this graphic, a

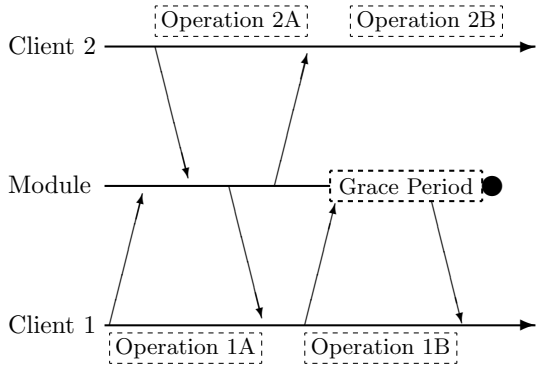

Figure 1: Graphical explanation of RCU and the concept of grace periods (time increases to the right)

Linux kernel module is being accessed by two readers ('Clients'). In this case, RCU protects the reader threads against module unloads. The slanted arrows represent invocations and responses from the module. To guard against race conditions, RCU uses the following strategy: The access operations performed by the client threads each form a so-called *read-side critical section*. The thread performing the unload now simply has to wait for the completion of all read-side critical sections that were entered before the start of the write operation. This allows ongoing operations to finish [7]. A read-side critical section is said to be *pre-existing* with respect to an update when its commencement falls before the start of the update. The time interval between the publication of a new version of a data element and the completion of the last pre-existing read-side critical section is referred to as *grace period*.

Any read access which started after the module unloading invocation, such as Operation 2B, is informed that the module is no longer available. Therefore, at the end of the grace period, it is guaranteed that no more readers have access to the data element in question and the module can safely be unmapped. As long as all read-side critical section have a finite duration, the grace period will always complete.

One final challenge remains: To implement RCU we need a reliable method to determine the end of the grace period. There are several possible strategies to accomplish this. The most well known method takes an indirect approach based on monitoring thread context switches. It requires read-side critical sections to be non-preemptable and non-blocking. In this approach, the end of the grace period has arrived when

each CPU has performed at least one context switch, since no context switches are allowed in read-side critical sections.

Let us look at an example application of RCU: Consider as an example the case of modifying an element of a single-linked list, a data structure frequently used within the Linux kernel. Using RCU, the writer thread first creates a copy of the element to be modified and then sets its data members to their new values. After that, the updater uses RCU's publication facilities to atomically adjust the next pointer of the previous element, guaranteeing that subsequent readers traversing the list can only see the modified element. Now the updating thread waits until all pre-existing readers that still access the old version of the list element have relinquished their pointers to it. When that has occurred, the grace period is over and the writing thread can free the old element.

It has to be emphasised that RCU's approach to synchronization is inherently pessimistic in nature. It is possible that an updating thread is impeded even though there no pre-existing read-side operations exist on the data element that is to be updated. The updater has to wait until *all* pre-existing read-side critical sections (on all data elements in the system) have been completed [7]. However, this strategy does not rely on atomic operations (such as reference counters for kernel modules). It drastically improves reader-side scalability because the execution overhead imposed by the synchronization primitives is constant.

2. RCU IN THE LINUX KERNEL

Traditionally, most of the use cases of RCU centered around synchronization within operating system kernels because these systems include many read-intensive data structures that profit highly from RCU's read-side performance. The first operating system to incorporate RCU was DYNIX/ptx 2.1 (released in 1993) [9]. However, the real breakthrough of RCU was its introduction into the Linux kernel in 2002 with kernel version 2.5.43 [11].

The oldest Linux systems with multiprocessor support used a big kernel lock (BKL), a single global lock that had to be acquired by each thread upon entering kernel space to serialize execution. This approach resulted in an unacceptable performance degradation on multi-core systems. Through the use of RCU and other techniques such as fine-grained locks and lock-free data structures, developers were able to greatly increase concurrency [8]. Today, RCU is present in almost every kernel subsystem. This section presents the design and implementation of Linux's RCU primitives, statistics for RCU usage within the kernel and example RCU applications.

2.1 Design of Linux RCU primitves

RCU is implemented as a library within the Linux kernel which includes the following basic primitives [8]:

- `rcu_read_lock/rcu_read_unlock`: Threads call these two functions upon entering or exiting a RCU read-side critical section, respectively.
- `synchronize_rcu`: Waits until all pre-existing read-side critical sections have completed but does not prevent new read-side critical sections from starting. There is also a non-blocking version of this function which asynchronously executes a callback once existing RCU critical sections have completed.

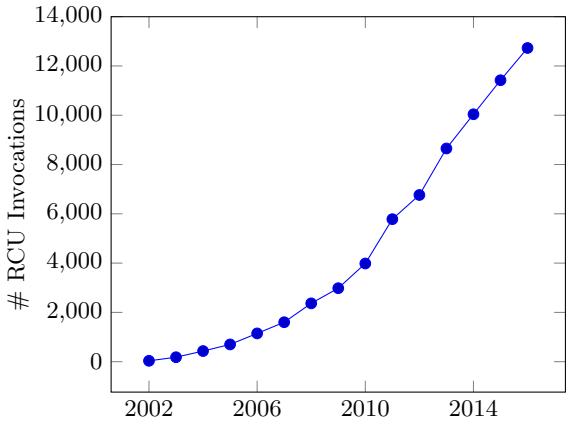

Figure 2: Usage statistics of the RCU API in the Linux kernel over time

- `rcu_dereference/rcu_assign_pointer`: These functions are necessary to to prevent that a reader gains access to a data element before it has been completely initialized by the writer. Together, these primitives enforce the required memory barrier instructions and compiler directives to ensure correct ordering when accessing pointers to data elements.

In addition to the functions above, the Linux RCU API also includes provisions for dealing with common use cases and data structures, such as reading or mutating the elements of a linked list.

To determine the end of grace periods, Linux uses context switches, as discussed in Section 1. One benefit of this approach is that is does not explicitly require communication between readers and callers of `synchronize_rcu`. Roughly speaking, Linux's RCU primitives are implemented as follows: In preemptable systems, on the one hand, a call to `rcu_read_lock` disables preemption on the calling CPU while `rcu_read_unlock` restores the preemption status. This satisfies the requirement that threads inside a read-side critical section may not context switch. Since the preemption flag is CPU-local, no race conditions exist when setting the variable.

For non-preemtable systems, on the other hand, the two delimiter functions generate no code at all, resulting in an optimal read-side performance. `synchronize_rcu` simply waits for all cores to execute a context switch.

Of course, the actual implementation is more intricate than the one sketched above since it also has to deal with interrupts, CPU hot-plugging and other hazards [10]. The implementation for real-time systems is also quite different. In principal, however, it can be stated that the Linux RCU design delays `synchronize_rcu` callers a little longer than necessary in an effort to reduce the overhead of read-side critical sections [8].

2.2 Linux RCU usage statistics

Since its introduction into the kernel, the usage of the RCU API facilities within Linux has skyrocketed (see Figure 2). As of December 2016, there are 12,729 invocations of RCU primitives within the kernel [6]. This increase can in part be attributed to the fact that many uses of read-write locks were replaced by RCU.

In 2012, McKenney et al. undertook a detailed investigation of the distribution of RCU API calls among the various kernel subsystems [8]. They found that the networking stack and the kernel-based virtual machine were the most intensive RCU users since they contain many read-intensive data structures that are especially suited to RCU (for example, the device configuration and routing within the networking system). The architecture-specific code (`arch`) uses RCU the least. This can in part be explained by the requirement to update hardware state in place. For example, it is not possible for updaters to create new versions of actual hardware registers while pre-existing readers access the old version.

However, if we compare RCU to other synchronization mechanism in the Linux kernel such as spinlocks and semaphores, we discover that RCU's usage only accounts for about 10 percent of calls to synchronization primitives. This can be attributed to the fact that RCU is a highly specialized technique for concurrent accesses to read-intensive data structures [6]. Furthermore, RCU still requires traditional locking to serialize multiple updaters.

2.3 RCU usage examples

In Section 1, we already illustrated the basic principles of RCU with an example from the Linux kernel, namely the dynamic unloading of kernel modules. We now look at another application of RCU within the Linux NMI (non-maskable interrupt) subsystem. NMIs are a special kind of hardware interrupts that can not be ignored using the normal interrupt masking facilities. They can signal non-recoverable hardware errors needing immediate attention. Additionally, they are often used for debugging purposes.

In Linux, it is possible to define handler functions for NMIs by installing them in the NMI handler table. RCU is used to protect the NMI handler table from concurrent updaters. In this case, the readers are CPUs that execute a NMI handler. When unregistering an NMI handler, it must be assured that its code remains intact as long as there are CPUs executing it in order to prevent access to invalid memory.

Figure 3 shows a pseudocode implementation of the Linux NMI system using RCU synchronization. Once on NMI occurs, the interrupted CPU calls `handle_NMI` in order to gain read access to the NMI handler table. Updaters make calls to `(un)register_nmi_handler` to manipulate the table and to remove and insert NMI handlers. The handler table is implemented as a double-linked list.

We know take a closer look at each function. Once an NMI has been delivered to a CPU, the resulting call to `handle_nmi` accesses the NMI handler table (`nmi_list`) in a read-side critical section (note the two delimiting functions). The `rcu_list_for_each` function is part of the RCU API. It iterates over each element of the double-linked list by calling `rcu_dereference` for each list element. Each NMI handler is then executed within a critical section by calling `cb` for each of them (`cb` stands for the callback function).

The other two functions defined in Figure 3 manipulate the list and allow to register and unregister NMI handlers. Updaters use the spinlock (`nmi_list_lock`) to serialize themselves. Both `rcu_list_add` and `rcu_list_remove` call `rcu_assign_pointer` to safely communicate the change of the pointer value to the reader (using the required memory order barriers and compiler directives). In addition, `unregister_nmi_handler` also invokes `synchronize_rcu` to

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;

void handle_nmi()
{
    rcu_read_lock();
    rcu_list_for_each(&nmi_list, handler_t cb)
        cb();
    rcu_read_unlock();
}

void register_nmi_handler(handler_t cb)
{
    spin_lock(&nmi_list_lock);
    rcu_list_add(&nmi_list, cb);
    spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb)
{
    spin_lock(&nmi_list_lock);
    rcu_list_remove(cb);
    spin_unlock(&nmi_list_lock);
    synchronize_rcu();
}
```

Figure 3: Usage of RCU to protect the NMI handler table

await the completion of pre-existing critical sections. Upon its return all calls to the handler have finished [8].

Implementing an NMI handler table in this fashion has multiple advantages. It guarantees high performance and using the RCU primitives imposes a low execution overhead since `rcu_read_lock`/`rcu_read_unlock` execute in a deterministic amount of time. In addition, the RCU approach is not prone to deadlocks, allowing the dynamic registration of NMI handlers which was not possible in previous kernel versions. Using a conventional locking technique, a thread holding the lock could be interrupted in `register_nmi_handler` by an NMI which would lead to a call to `handle_nmi`. This function would also attempt to acquire the lock, leading to a deadlock.

Several other applications for RCU within Linux have been described in various research papers, including the Virtual File System (VFS) dentry cache [8] or the System V IPC mechanism which uses RCU as an alternative to read-write locks [2].

3. DISCUSSION OF RCU

This section presents the concurrency situations for which RCU is best suited and also addresses some of the problems with using RCU that may lead to a different choice of synchronization mechanism.

Since RCU's distinguishing property is the support for concurrent readers in the presence of updaters, RCU can lead to massive performance gains on read-intensive data structures. McKenney et al. [7] have provided a rule of thumb: A data structure is read-intensive if

$$f \ll \frac{1}{n_{CPU}}$$

holds where f is ratio of updates to total acesses and n_{CPU} is the number of CPUs in the system. With eight CPUs,

f should be far less than 0.125. In general, RCU is inadequate for data structures with a high number of modifications since it might delay updaters far longer then necessary. A RCU grace period can extend for multiple milliseconds which might be intolerably high for some applications [4]. In addition, RCU should not be used for CPU-bound code [7].

A big advantage of RCU is its low storage overhead. For example, the Linux RCU provisions for delimiting critical sections do not introduce any variables at all. The execution overhead for RCU is also quite miniscule (or even inexistent in the case of non-preemptive Linux systems), and even more importantly, deterministic. This is especially beneficial for real-time systems.

Additionally, the execution overhead for RCU critical sections remains constant even if the number of CPUs increases. The relatively high scalability of RCU is a great improvement over conventional locking strategies [8].

One drawback of RCU is the fact that readers might have access to stale data. When an updater has modified a list element, pre-existing readers still see the old state. An application developer must determine if this behaviour can be tolerated. One approach is the use of a flag to identify outdated information, allowing a reader to take appropriate measures.

Another disadvantage of RCU is the fact that it requires multiple version of data elements to be maintained in case an update occurs. Depending on the size of a data element, this can considerably increase the memory overhead. To reduce memory consumption, is is possible to limit the maximum number of versions of data structure by letting the writer threads hold the semaphore protecting the updates during the grace period [10].

Finally, we should examine if RCU adheres to task priorities which is especially important for designers of real-time systems. Here, we can also see that no strength comes without a weakness: On the one hand, RCU is immune to priority inversions involving read-side primitives. On the other hand, a high-priority process can be blocked by low-priority readers waiting for the elapse of a grace period [4].

In summary, RCU can best play out its strengths when used in conjunction with read-mostly data structures since RCU favours readers over writers. Additional benefits of RCU are the low storage overhead and the obedience to reader task priorities. For some applications, however, RCU can be an ill-suited choice. The include update-intensive data structures or applications that cannot tolerate the momentary inconsistencies imposed by RCU updates.

4. USER-LEVEL RCU

Though RCU has been mostly used for synchronization within operating system kernels in the past, various researchers have undertook efforts to make the benefits of RCU available to user-level applications as well. However, developers face several challenges when attempting to implement RCU primitives outside the kernel.

One problem is that most kernel-level implementations assume that threads cannot be preempted while executing a read-side critical section. This assumption does not hold in userspace. Making kernel RCU primitives directly available to user applications is also not a good option for various reasons. For example, it has to be prevented that a thread can hang the entire system by staying in a critical section indefinitely. In addition, the induced system call overhead would render all performance benefits of RCU void [3].

McKenney et al. addressed these problems in their userspace RCU library (URCU) which was released in 2009 and is now available in several Linux distributions (in Debian, for example, as `liburcu2`) [5]. This library lets the user choose between several different implementations of RCU (called 'flavors') depending on the requirements of his application. Each of them tackles the challenges involved in making RCU available in userspace in a different way. Currently, URCU includes the following implementations:

- *quiescent-state-based RCU (QSBR)*: Like the Linux kernel implementation, QSBR provides almost zero-cost reads. However, it requires threads to periodically announce that they are in a quiescent state. QSBR uses these announcements to approximate the length of read-side critical section. This approach severely restricts application design, making it impossible to use within libraries.

- *Memory-barrier-based RCU*: This is a general purpose RCU implementation that can be used in almost any software environment since it does not require quiescent state announcements. However, it imposes a higher read-side overhead because the `rcu_read_(un)lock` primitives contain memory barriers. It also requires readers to track the nesting of critical sections. Conceptually, the implementation divides grace periods into two different *phases*, with phase change being initiated when a writer thread calls `synchronize_rcu`. Reader threads save a snapshot of the current phase value upon entering an outermost RCU critical section.

- *Signal-based RCU*: In this approach, the updater sends POSIX signals to the reader threads. It therefore requires reserving one signal for RCU. However, the implementation can dispense with memory barriers in read-side primitives, making reads almost as fast as with QSBR. Additionally, the signal-based approach compels threads to register themselves before entering their first read-side critical section.

When selecting the right 'flavor' for his application, a developer must therefore perform a trade-off between optimizing the performance of read operations and imposing a minimum amount of changes to existing code.

Several possible application scenarios for RCU have already been proposed, including the LTTng tracer or the BIND domain-name server. LTTng [1] is a userspace tracer that carries out performance analysis and assesses the interaction between userspace applications and the Linux kernel. Since it cannot enforce changes to existing user-level applications, it cannot use the QSBR approach discussed above. However, the memory-barrier-based implementation is an option.

Another use case for URCU is BIND, which is the de-facto standard for DNS software on the Internet. BIND is encountering suboptimal scalability with multiple processors. Since domain names are often read but seldom updated, user-level RCU implementation could improve performance. Others have suggested benefits of RCU for financial applications [3].

5. CONCLUSION

RCU is a synchronization mechanism designed to optimize the performance of read operations. It provides wait-free and low overhead reads with deterministic execution time. These beneficial properties go to the expanse of writers which can be delayed for a considerable period of time. Closely related to RCU is the concept of grace periods which ensures that a version of a data structure remains intact as long as it is accessed by readers.

RCU is widely used in the Linux kernel in various subsystems since many internal kernel data structures are read-intensive. More recently, a userlevel RCU implementation was developed to broaden the range of possible RCU applications.

The papers of Paul E. McKenney [7] [8] provide a comprehensive survey of the usage scenarios of RCU and discuss the different RCU implementations. They also include detailed performance comparisons.

6. REFERENCES

[1] Lttng: an open source tracing framework for linux. Available: http://www.lttng.org/, 2017.

[2] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 297–310. USENIX Association, June 2003. Available: http://www.rdrop.com/users/paulmck/RCU/rcu.FREENIX.2003.06.14.pdf.

[3] M. Desnoyers, P. E. McKenney, A. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.

[4] P. E. McKenney. What is RCU? part 2: Usage. Available: http://lwn.net/Articles/263130/, January 2008.

[5] P. E. McKenney. User-space rcu. Available: https://lwn.net/Articles/573424/, November 2013.

[6] P. E. McKenney. Rcu linux usage. Available: http://www.rdrop.com/~paulmck/RCU/linuxusage.html/, 2017.

[7] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001. Available: http://www.linuxsymposium.org/2001/abstracts/readcopy.php http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf.

[8] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU usage in the linux kernel: One decade later. Technical report paulmck.2012.09.17, September 2012.

[9] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998. Available: http://www.rdrop.com/users/paulmck/RCU/rclockpdcsproof.pdf.

[10] P. E. McKenney and J. Walpole. What is RCU, fundamentally? Available: http://lwn.net/Articles/262464/, December 2007.

[11] L. Torvalds. Summary of changes from v2.5.42 to v2.5.43. Available: https://www.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.43/, 2002.