A Portable Read-Copy-Update Algorithm

Post-Graduate Student:
MGR. ANDREJ PODZIMEK

Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ

podzimek@cs.cas.cz

Supervisor:
ING. LUBOMÍR BULEJ, PH.D.

Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ

bulej@cs.cas.cz

Field of Study:
Software Systems

Abstract

RCU is a synchronization mechanism that can increase concurrency in parallel algorithms, improving scalability in comparison to mutual exclusion. RCU provides asymmetric synchronization of concurrent writers and readers sharing a data structure. Unlike mutual exclusion primitives, RCU can avoid expensive memory operations on the most frequent code paths, boosting performance even on uniprocessors. Virtually all contemporary RCU implementations run in the Linux kernel and strongly depend on its internals.

Our work contributes a novel RCU algorithm based on easily portable foundations, not bound to any particular kernel architecture. We implemented and benchmarked our algorithm in the UTS kernel used by Solaris-based systems. We compared our RCU algorithm to a readers-writer lock and to a portable, but feature-constrained RCU algorithm called QRCU. Our benchmarks suggest that the novel algorithm can outperform both readers-writer locks and QRCU on current SMP systems.

1. RCU Essentials

Read-Copy-Update (RCU) is a means of communication among three types of entities: readers, writers and reclaimers [1].

Readers access a shared data structure without modifying it and can run in parallel with other readers and writers, guaranteed to never block when entering or leaving their critical sections. Most RCU implementations do not require the readers to use atomic instructions or other expensive operations.

Writers are a specific type of readers that can also modify the shared data. Writers cooperate with the RCU mechanism to provide other readers with an illusion of data integrity, i.e. readers will not observe concurrent changes to the shared data during their critical sections. This is achieved by copying the shared data structure, making changes to the copy and finally replacing the pointer to the original data structure with a pointer to the new one atomically. As long as readers adhere to certain data access rules, they always observe a consistent state of the data structure. RCU neither supports nor constrains concurrency among writers; they have to synchronize their operations by means external to RCU.

Deallocation of old versions of protected data has to be postponed, so that readers accessing them can finish their work. The time needed for all potential readers to stop using the old data structure (no longer accessible to new emerging readers) is called a *grace period*. A moment when a potential reader does not access any data structure protected by RCU is called a *quiescent state*. A grace period elapses when all potential readers go through at least one quiescent state. Grace period detection is the key part of all RCU implementations.

Reclaimers deallocate outdated data structures that had been made inaccessible to readers. It is necessary to wait for at least one grace period before the deallocation can be done. Writers can use the RCU mechanism to block for at least one grace period, becoming reclaimers afterwards. Alternatively, they can proceed immediately, asking the RCU mechanism to perform the deallocation when appropriate. Our novel RCU algorithm supports both of these options.

2. The RCU Algorithm for UTS

The cornerstone RCU algorithms in the Linux kernel are strongly bound to features specific to Linux, such as timer interrupt handling on all processors. In the UTS kernel, timer interrupts are only handled by a subset of available processors, which may only include one processor on UMA machines [2]. This fundamental difference makes porting of the key Linux RCU algorithms to UTS or other kernels technically infeasible. The design of our novel RCU algorithm strives to avoid technical dependencies related to one particular kernel.

The key idea behind our algorithm can be illustrated on a "toy" RCU algorithm presented by Paul McKenney [3]: Writers context-switch themselves to each available processor before they become reclaimers. Since readers run with disabled preemption, the writers' behavior guarantees that at least one grace period must have elapsed. Presumably, this algorithm is unusable in practice. First, its SMP scalability would be extremely poor. Second, it does not support non-blocking writers and delayed batched resource reclamation.

Based on the principle mentioned above, we designed a more scalable algorithm where forced rescheduling is only used as the last resort when other means of grace period detection take too long to complete. Our novel algorithm differs from the trivial example above in a number of ways. First, all grace period detection requests are batched and handled centrally by one detector thread, which avoids the need to reschedule each writer on each processor on each request. Second, the central detector thread avoids forced migration in most cases, at the cost of slightly higher overhead on the readers' side. Third, most of the advanced RCU features, such as asynchronous reclamation, are implemented.

A brief note on notable characteristics of our algorithm follows. Readers do not use any expensive atomic instructions. Readers only execute memory barriers when intensive grace period detection takes place; they never do so in the absence of grace period requests. Naturally occurring quiescent states (context switches, idle processors) are observed to reduce the grace period detection overhead even further. As long as all read-side critical sections take a bounded amount of time (which can be required and relied upon in a kernel environment), grace period duration is also bounded. Asynchronous reclamation requests are handled in efficient batches by the same processor on which they were created, so that a warm cache can be exploited.

3. Evaluation

To verify that our RCU algorithm for the UTS kernel leads to performance improvements typical for well-known RCU implementations, we created a benchmarking harness that performs a series of operations on a non-blocking hash table. This artificial workload simulates a kernel algorithm manipulating a data mapping under heavy stress. The same workload (sequences of hash table operations performed by multiple threads in parallel) has been benchmarked with four different synchronization mechanisms protecting the hash table. We ran our benchmark on a variety of SPARCv9 and x86-64 SMP machines.

	511:1	127:1	31:1	7:1	1:1
RCUc	**1**	1.04	1.06	1.12	1.27
RCUs	1.03	1.23	1.48	2.23	5.24
QRCU	2.33	2.33	2.47	3.06	4.55
DRCU	2.86	4.21	8.95	N/A	N/A

Table 1: Relative average running time

Selected benchmark results (from an x86-64 machine with 8 processors) are shown in Table 1. Relative running times of our multithreaded workload are displayed, normalized so that the shortest measured result takes one time unit. Columns represent ratios between frequencies of read-only and read/write operations on the hash table. Rows represent synchronization mechanisms. RCUs and RCUc denote our algorithm with its synchronous and asynchronous reclamation handling API, respectively. QRCU denotes the feature-constrained RCU algorithm [4] ported for the sake of comparison. DRCU ("dummy RCU") stands for an implementation of the RCU API using a plain readers-writer lock.

Since RCU is designed for read-mostly workloads, significant improvements over DRCU under high readers/writers ratios are not surprising. Interestingly, our novel algorithm performed relatively well even under low readers/writers ratios.

References

[1] P. E. McKenney, *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. http://www.rdrop.com/users/paulmck/RCU/RCU dissertation.2004.07.14e1.pdf.

[2] R. McDougall and J. Mauro, *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Solaris Series, Sun Microsystems Press/Prentice Hall, 2007.

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

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