Poor Man's URCU

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Abstract

RCU is, among other things, a well known mechanism for memory reclamation that is meant to be used in languages without an automatic Garbage Collector, unfortunately, it requires support in the kernel or operating system, which is currently provided only in Linux. An alternative is to use Userspace RCU (URCU) which has two main variants that can deployed on other operating systems, named *Memory Barrier* and *Bullet Proof*.

We present a novel algorithm that implements the three core APIs of RCU: `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()`. Our algorithm uses one mutual exclusion lock and two reader-writer locks with `trylock()` capabilities, which means it does not need a language with a memory model or atomics API, and as such, it can be easily implemented in almost any language, regardless of the underlying CPU architecture, or operating system.

Categories and Subject Descriptors D.4.1.f [*Operating Systems*]: Synchronization

General Terms RCU, locks

Keywords RCU, locks

1. Introduction

When implementing lock-free and wait-free data structures in languages without an automatic Garbage Collector (GC), or just to manage object lifetime, some kind of memory reclamation technique is required. Several such techniques have been discovered [1, 2, 4, 5, 8, 9] with RCU [4, 8] one of the most well known. RCU has two particularly interesting properties for its non-reclaiming methods, i.e. `rcu_read_lock()` and `rcu_read_unlock()`, one being its wait-free progress condition, and the other its low impact on latency with high scalability. RCU is unfortunately not a completely generic technique because it requires support in the kernel or operating system.

Fortunately, there is Userspace RCU (URCU) [4] which has two variants that are generic enough to be implemented regardless of operating system support or other library APIs, named *Memory Barrier URCU* and *Bullet Proof URCU*. Even so, these two variants of URCU are currently only implemented in C/C++ and although many architectures are supported, the support is not universal, and other languages may have difficulty linking with the library.

We present a new algorithm which we named *Poor Man's URCU* which can be easily implemented in any language, as long as the language provides a mutual exclusion lock and a reader-writer lock with `trylock()` functionality. This means our algorithm can be implemented across a large number of languages, without need for a memory model, or atomics API, and regardless of the operating system on which it runs. Our algorithm implements only the core functionality of RCU, namely `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()`, and it does so providing only lock-free progress for the first two, while RCU and URCU tipically provide wait-free progress.

2. Algorithm

The algorithm used in *Poor Man's URCU* is composed of one mutual exclusion lock and two reader-writer lock instances.

Threads that call `rcu_read_lock()` and `rcu_read_unlock()` (Arrivers, or Readers) will attempt to acquire the reader lock on the first of the reader-writer locks (`rwlock1`) with a `trylock()` operation, and if it fails, do a `trylock()` on the second reader-writer lock (`rwlock2`), and if that also fails, then try again the first rwlock, and so on. This procedure could go on indefinitely, but it should only fail if there is another thread doing progress, namely, a thread acquiring the write-lock in one of the reader-writer locks, which means the operation is lock-free.

A thread that calls `synchronize_rcu()` (Toggler or Writer) will start by acquiring the lock on the mutual exclusion lock to guarantee only one Toggler at a time is present. The Toggler will then acquire the write-lock on the second reader-writer lock and release it as soon as it has obtained it, followed by acquiring the write-lock on the first reader-writer lock and release it as soon as it has obtained it. This is enough to introduce a linearization point [6], and it is now safe for the Toggler to free/delete the memory of the object that is meant to be reclaimed (or whatever other task it needs to do), because it has the guarantee that any ongoing Readers can no longer hold a pointer to such object.

Algorithm 1 shows the implementation in C99 of *Poor Man's URCU* using the Pthreads library API. Notice that in this implementation the `rcu_read_lock()` API returns which reader-writer lock was acquired in the form of an integer, which is then passed to `rcu_read_unlock()` as an argument, so that it knows which reader-writer lock to unlock, but an alternative implementation could be done using a thread-local variable to pass this information. An example of how to use this API for memory reclamation can be seen here [7].

Although in our implementation the order of acquiring the lock on `synchronize_rcu()` is first on `rwlock2` and then on `rwlock1`, it would be equally correct to lock/unlock in the opposite order.

3. Performance

Figure 1 shows the mean for 5 runs of the number of operations per second, per number of threads, on a microbenchmark that iterates for 20 seconds. Each read-iteration calls `rcu_read_lock()`, reads all items in an array of 100 user-defined objects (read-critical section), then calls `rcu_read_unlock()`. Each write-iteration creates a new user-defined object, replaces a random object in the array of 100 using `compare_exchange_strong()`, calls `synchronize_rcu()`, and then it safely `frees` the memory of the previous instance because the RCU mechanism provides the guarantee that there is no thread currently holding a reference to that particular object. The benchmark was ran on a PowerPC

Algorithm 1 Pthread implementation of Poor Man's URCU

```
1  typedef struct {
2    pthread_mutex_t  togglerMutex;
3    pthread_rwlock_t rwlock1;
4    pthread_rwlock_t rwlock2;
5  } poorman_urcu_t;
6
7  // The return value should be passed as arg 'whichone'
8  int rcu_read_lock(poorman_urcu_t * self) {
9    while (1) {
10     if (pthread_rwlock_tryrdlock(&self->rwlock1) == 0) {
11       return 1;
12     }
13     if (pthread_rwlock_tryrdlock(&self->rwlock2) == 0) {
14       return 2;
15     }
16   }
17 }
18
19 void rcu_read_unlock(poorman_urcu_t * self, int whichone) {
20   if (whichone == 1) {
21     pthread_rwlock_unlock(&self->rwlock1);
22   } else {     // whichone == 2
23     pthread_rwlock_unlock(&self->rwlock2);
24   }
25 }
26
27 void synchronize_rcu(poorman_urcu_t * self) {
28   pthread_mutex_lock(&self->togglerMutex);
29   pthread_rwlock_wrlock(&self->rwlock2);
30   pthread_rwlock_unlock(&self->rwlock2);
31   pthread_rwlock_wrlock(&self->rwlock1);
32   pthread_rwlock_unlock(&self->rwlock1);
33   pthread_mutex_unlock(&self->togglerMutex);
34 }
```

Power8E with 8 cores, running Ubuntu Linux 14.04 64 bit, with GCC 4.9.2. The Memory Barrier implementation of URCU requires prior thread registration, but the Bullet Proof does not, therefore, in order to have a fair comparison with Poor Man's URCU which also requires no registration of threads, we compared only against the Bullet Proof implementation, as shown in Figure 1.

Although there is a performance advantage of the Bullet Proof algorithm, the Poor Man's implementation is not that far behind and it surpasses in the special case of 1 thread at 100% (the thread is doing only calls to `synchronize_rcu()`). We should keep in mind that the performance of Poor Man's URCU is tightly coupled to the underlying implementation of the Reader-Writer Locks used, and recent advances in this area [3] should provide significant improvement.

4. Conclusion

Although not particularly efficient, and with a limited API, our *Poor Man's URCU* algorithm is highly portable across Operating Systems and programming languages, does not require linking with an external library, does not require kernel/OS support, and has no licensing constraint. The simplicity and ease-of-use of our algorithm can foster the usage of RCU in domains where before it was impractical.

References

[1] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, page 25. ACM, 2014.

[2] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 33–42. ACM, 2013.

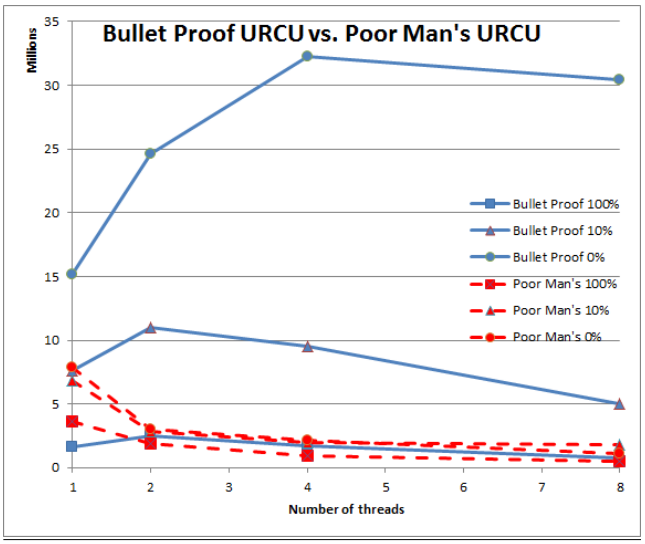

Figure 1. The plot above shows the total number of operations as a function of the number of threads, on a PowerPC machine with 8 cores. The `100%` lines means that each thread would only call `synchronize_rcu()`. The `10%` lines means that each thread would call `synchronize_rcu()` 10% of the times, and the remaining 90% of the times it would call `rcu_read_lock()` followed by `rcu_read_unlock()`.

[3] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. *PPoPP 2013*, 2013.

[4] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):375–382, 2012.

[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[6] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[7] P. E. McKenney and J. Walpole. What is rcu, fundamentally? 2007.

[8] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[9] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.