UNIVERSITY OF WATERLOO
Faculty of Mathematics

RCU Hash Table

Blackberry Limited
CoreOS Team
Ottawa, Ontario, Canada

Prepared by
Zhaocheng Che
20818599
4A Computer Science
December 4, 2021

Letter of Submittal

To: David Sarrazin
From: Zhaocheng Che
Date: December 14, 2021
Re: Work Report: RCU Hash Table

This report entitled "RCU Hash Table" was created as my 4A Work Report for the CoreOS Team of Blackberry Limited. This work report is the second of the four work reports I am required to complete as part of my Bachelor of CS Co-op degree requirement. The purpose of this report is to explore the Read-copy update (RCU) mechanism and propose a concurrent hash table based on it.

The CoreOS team, in which I am employed, is devoted to developing the kernel of the realtime operating system QNX. QNX is targeted at the embedded systems such as automobiles. My main duty is writing unit-tests for the file systems module.

The Faculty of Mathematics requests that you evaluate this report for command of topic and technical content/analysis. Following your assessment, the report, together with your evaluation, will be submitted to the Math Undergraduate Office for on-campus evaluation by qualified work report markers. The combined marks determine whether the report will receive credit and whether it will be considered for an award.

To my best knowledge, I certify that the intellectual content of this report is the product of my own work and that all the assistance and sources are acknowledged.

Thank you,

Zhaocheng Che

Table of Contents

List of Figures

Executive Summary

The purpose of the report is to implement a concurrent hash table using the RCU mechanism.

In the beginning, an introduction about RCU describes the core ideas and the APIs in the Linux kernel. The concept motivates the design of the proposed hash table. Simplified implementation in C is included for illustration, and the details are discussed to explain its correctness, especially of the concurrent operations.

In the end, a performance test is conducted on the "single mutex" hash table and the RCU hash table. The latter dominates in all test configurations and demonstrates promising scalability in terms of the number of parallel operations.

1.0 Introduction

Concurrent data structures are intended to be shared among multiple threads and manage their parallel accesses. Compared to the sequential data structures, it adds an extra level of complexity: the interleaving of the instructions from different threads. There could be countless possible interleaving orders, and the designer must only allow those orders with which all methods will obtain correct results defined by the data structure.

The simplest method to guarantee correctness is to mandate no overlaps between any two data structure accesses. However, when the number of parallel operations is high, some threads need to wait a long time for the previous operations to finish. The penalty will even grow with the number of concurrent operations. Since the multi-threading ability of modern CPU is increasing, many strategies are designed to decrease this penalty, and Read-copy update (RCU) is one of them.

An OS kernel provides system calls that serve all running threads on this OS. The number of running threads is determined by the CPU. Therefore, the kernel's concurrent performance should be optimized for use on multi-threaded CPUs. Hash table is a common data structure in the Linux kernel. This report applies the RCU synchronization mechanism to build a concurrent hash table and analyzes its performance with a experiment.

2.0 Analysis

2.1 RCU Background

Read-copy update (RCU) is a synchronization mechanism supporting concurrency between a single updater and multiple readers. The design distributes the work between read and update paths in such a way as to make read paths extremely fast.

The core of the RCU design is to divide the process of the writer into three phases: update, Grace Period and reclamation. A typical RCU writer sequence goes like the following (McKenney, n.d.):

1. Update/remove pointers to a data structure, so that subsequent readers cannot gain a reference to the old one,
2. Wait for all previous readers to complete their RCU read-side critical sections, this period of time is called the Grace Period.
3. At this point, there cannot be any readers holding references to the data structure, so it now may safely be reclaimed (e.g., kfree()).

Usually, most of computation in the writer is done in the update phase, and only the task of freeing resource is required to stay in the reclamation phase. The meaningful result is that the update phase doesn't actually need to know anything about concurrent readers. This unburdens the readers and writers to communicate with each other during the heavier computation.

Furthermore, in the Grace Period, the writer only needs to wait for all **previous** readers to finish the RCU read-side critical sections; it doesn't block new readers from entering. In fact, in RCU, nothing will block the readers from entering and exiting the critical section.

In addition, overhead of the RCU read-side cirtical section is also very low, in some cases, absolutely no synchronization at all. (McKenney, n.d.). In contrast, in more conventional lock-based strategies, readers must use heavy-weight synchronization in order to prevent an updater from deleting the data structure out from under them. In RCU, the task is given to the writer's Grace Period instead. Thus, RCU is very suitable for read-mostly data structures.

One of the simplest implementation for the Grace Period is based on disabling preemption. (McKenney et al., 2002) Similar to holding a spinlock, this implementation mandates no context switch inside the read-side critical section. Therefore, after a CPU having been seen performing a context switch, it cannot possibly still be inside a read-side critical section entered before the context switch. Thus, once the writer updates the data pointer and it sees all CPUs having done a context switch, the old data can be freed. (There are much more efficient implementations descirbed elsewhere.) (McKenney et al., 2002)

```
rcu_read_lock // enter read-side critical section
rcu_read_unlock // exit read-side critical section
rcu_dereference // obtain a copy of the shared pointer
rcu_assign_pointer // update the value of the shared pointer
synchronize_rcu // blocks during Grace Period
call_rcu // callback form of synchronize_rcu
```

("Linux source code", 2021, `include/linux/rcupdate.h`, Line 683)

It is important to note that the readers to the shared pointer should use the given API `rcu_dereference` and the writers should use `rcu_assign_pointer` instead of `=`. These two APIs will prevent the compiler and the CPU to reorder the code. (McKenney & Walpole, 2007)

There are two APIs for the reclamation phase: `synchronize_rcu` and `call_rcu`.

`synchronize_rcu` marks the end of updater code and the beginning of reclaimer code. It blocks until all pre-existing RCU read-side critical sections on all CPUs have completed. ("What is RCU? – Read, Copy, Update", n.d.) `call_rcu` is a callback form of `synchronize_rcu`. It will register a function and arguments, and invoke that in another thread after the Grace Period. (Arcangeli et al., 2003) `kfree` is such a common reclamation phase function that there is a speicial shortcut `kfree_rcu` which is equivalent to `call_rcu` with `kfree`.

An example of readers/writers of the shared data pointer is

```
void reader(void) {
    rcu_read_lock();
    struct data * tmp = rcu_dereference(shared_pointer);
    // read tuple
    rcu_read_unlock();
}

void writer(void) {
    global_lock(); // exclusion for writers
    struct data * newdata = kmalloc(sizeof(struct tuple));
    struct data * olddata = shared_pointer;
    *newdata = *shared_pointer;
    // modify newdata
    rcu_assign_pointer(shared_pointer, newdata);
    kfree_rcu(olddata);
    global_unlock();
}
```

2.2 Linked List in Linux

The Linux kernel provides a commonly used linked list implementation: `hlist`.

The kernel defines two structs: `struct hlist_head` and `struct hlist_node`. ("Linux source code", 2021, `include/linux/types.h`, Line 182) And it also provides some functions and macros to manipulate the linked list:

```
void hlist_add_head(struct hlist_node *n, struct hlist_head *h);
void hlist_del(struct hlist_node *n);
hlist_for_each_entry(pos, head, member)
```

("Linux source code", 2021, `include/linux/list.h`, Line 879)

The usages are quite straight forward. A special RCU version of the previous functions and macros is also available:

```
void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);
void hlist_del_rcu(struct hlist_node *n);
hlist_for_each_entry_rcu(pos, head, member)
```

("Linux source code", 2021, `include/linux/rculist.h`, Line 584)

The difference is that `hlist_del_rcu` allows concurrent readers holding it to be able to access the next object in the list even after the node is removed; and `hlist_add_head_rcu` and `hlist_for_each_entry_rcu` use `rcu_assign_pointer` and `rcu_dereference` properly to handle concurrency. (McKenney & Walpole, 2007)

2.3 Hash Table

Hash tables are frequently used in the kernel code, often as a cache to speed up look-up operations. This report focuses on the implementation of the linked-list based concurrent hash table inside the Linux kernel. A possible C struct can be defined as below:

```
struct hashtable {
    struct hlist_head buckets[NUM_BUCKETS];
};

struct object {
    uint32_t o_hash;
    struct id o_id; // the unique id
    struct hlist_node o_node;
    spinlock_t o_lock; // RCU : protect the object
    int o_invalid; // RCU : denote the object is removed
    struct rcu_head o_rh; // kfree_rcu needs this field
    // data
};

struct mutex global_mutex;
```

Some common operations on the hash table are 1) insert an object; 2) remove an object; 3) lookup an object for read and write.

This report will focus on the concurrency among these operations. It should on one hand coordinate threads walking and modifying the hash lists, on the other hand protect reads and writes to the stored objects.

2.4 Single Mutex

The simplest approach is to use a single global mutex for everything. Each operation will execute with exclusive access to the hash table and all of the objects contained in it.

The implementation is almost identical to a single threaded hash table, with the exception of holding the global mutex during the operations. The report will briefly go over the implementation to highlight the difference between this and the RCU implementation in the next section.

1. Look-up

   The caller to the look-up operation must hold the global mutex. This function will compute the hash from the given id, find the corresponding bucket list head, and try to compare it with every entry in the list. It will return the object pointer if found, and NULL otherwise. The caller then will be able to use that pointer until it uses the pointer no longer and releases the global mutex.

2. Remove

   Holding the global mutex, `remove` will first call `lookup` for the given id to acquire the pointer to the object, then remove it from the list.

3. Insert

   Holding the global mutex, `insert` will first call `lookup` to ensure that there are no existing object with the same id, then it inserts the object to the head of the bucket's list.

As explained before, by guarding every operation with the global mutex, the operations on the hash table will always execute linearly. Therefore, the implementation doesn't scale with the number of parallel operations.

2.5 RCU locking

Some potential improvements are: 1) operations on different objects do not interfere with each other, so they can run in parallel; 2) read-only walks of the hash lists are also safe to run in parallel.

Thus, in the proposed RCU hash table, the responsibility of the big mutex is divided into two parts: a per-object spinlock for exclusive access to the object, and the RCU for protecting the hash lists. Therefore, threads only need to hold the RCU lock when walking the hash list and only the per-object lock when operating on a specific object. The finer control over the critical sections allows more parts of the program to run at the same time, thus improving the concurrent performance. The implementation and the effects are explained below.

2.5.1 Look-up

```
struct object * lookup_rcu(struct id *searchid) {
    // compute hash and bucket from searchid
    rcu_read_lock();
    struct object * objp;
    hlist_for_each_entry_rcu(objp, bucket, o_node) {
        if ((objp->o_hash == searchhash)
                && isequal(&objp->o_id, searchid)) {
            spin_lock(objp->o_lock);
            if (objp->o_invalid) {
                spin_unlock(objp->o_lock);
                rcu_read_unlock();
                return NULL;
            }
```

```
            rcu_read_unlock();
            return objp;
        }
    }
    rcu_read_unlock();
    return NULL;
}
```

The function enters the RCU read-side critical section to walk the hash list. RCU will protect the `objp` from being deleted, so it is always safe to access its content, including the `o_lock` and `o_invalid`. If an object in the list matches the search id, then obtain its `o_lock` to check the `o_invalid` flag. If the flag is set, then the function will return to the caller as not found. In the end, the function exits the RCU read-side critical section but will continue to hold the per-object `o_lock` for the caller if an object is returned. The design of the `remove_rcu` will ensure that the object will not be freed as long as `o_lock` is held.

2.5.2 Remove

```
void remove_rcu(struct id * removeid) {
    // compute hash and bucket from removeid
    mutex_lock(&global_mutex);
    struct object * objp = lookup(removeid);
    if (objp == NULL) {
        mutex_unlock(&global_mutex);
        return;
    }
    hlist_del_rcu(&objp->o_node);
    mutex_unlock(&global_mutex);
```

```
    spin_lock(&objp->o_lock);
    objp->o_invalid = true;
    spin_unlock(&objp->o_lock);
    kfree_rcu(objp, o_rh);
}
```

The function starts with obtaining the global mutex to look up the list. Since the list is not subject to change while the global mutex is held by this thread, the normal version of the lookup can be used. If the object is found in the hash table, then do the RCU version `hlist_del_rcu` to accommodate for the concurrent lookups. Then the global mutex can be released, since at this point, the object is not findable by any new hash table look-ups, so other threads `remove_rcu` and `insert_rcu` can start to execute with that premise. For example, if two removals are invoked on the same id, then the latter one to obtain the global mutex will find no object in the hash table.

Besides removals and insertions, concurrent `lookup_rcu`s must also be considered.

Firstly, by the previous explanation about the Grace Period, `lookup_rcu` can safely dereference the gained object pointer within the RCU read-side critical section. However, the returned reference from `lookup_rcu` is not protected by RCU anymore, it is only protected by `o_lock`, which will by itself guarantee the object's existence.

The key is that `lookup_rcu` obtains `o_lock` and checks `o_invalid` before it releases `rcu_read_unlock()`. This yields two possibilities: 1) `lookup_rcu` obtains `o_lock` before `remove_rcu`; 2) `lookup_rcu` obtains `o_lock` after `remove_rcu`.

In the first case, `remove_rcu` blocks until when the caller of `lookup_rcu` decides to release the `o_lock`, which implies that the object is no longer referenced. Therefore after that, `remove_rcu` can proceed to free the object.

In the second case, `lookup_rcu` will observe that `o_invalid` was set by `remove_rcu`; this happens inside the read-side critical section, so `remove_rcu` is kept in the Grace Period. Then, `lookup_rcu` releases the lock, exits the RCU read-side critical section, and returns `NULL`, only after which `remove_rcu` will proceed to free the object if the Grace Period ends. Therefore `o_lock` itself is enough to protect the existence of the object returned by `lookup_rcu`.

2.5.3 Insert

```
void insert_rcu(struct object * objp) {
    // compute the hash and bucket from object id
    mutex_lock(&global_mutex);
    hlist_add_head_rcu(&objp->o_node, bucket);
    mutex_unlock(&global_mutex);
}
```

The function's logic is the same as that of the normal version because it is essentially an atomic operation in the eyes of concurrent lookups. Because RCU lists only support walking the list in the forward direction (McKenney, 2007), the moment when searchers begin to see the added object is when the list head (a pointer) is updated.

2.5.4 Read and Write

```
struct object * objp = lookup_rcu(id);
if (objp != NULL) {
    // read/modify the object
    spin_unlock(objp->o_lock);
}
```

This piece of code follows directly from the explanation for `lookup_rcu` and `remove_rcu`.

2.6 Performance Comparison

2.6.1 Experiment

Two concurrent hash tables are implemented in the Linux kernel as a kernel module. The first hash table uses "a global mutex"; the second one uses the proposed RCU implementation.

To evaluate the concurrent performance of the hash tables, a certain number of threads (workers) are created and each runs an equal number of hash table operations in a loop (5% insertion, 5% removal, and 90% per-object operations in random order). The average time used in each thread is taken to approximate the total time required for the sum of the given operations on the workers; from which the number of operations per unit of time, or the processing speed, can be calculated. The experiment focuses on the correlation between the number of workers and the processing speed for each locking strategy.

In addition, the average length of the hash lists and the execution time of reading and writing an object are also controlled parameters.

The source code for the experiment can be downloaded at https://github.com/Darwin-Che/rcu_hashtable.

2.6.2 Result

The experiment was executed on a Ubuntu Server 20.04 in VMWare Fusion with 10 cores of Intel i9-9880H. Each test case is executed for about 5 seconds to allow the threads to obtain a stable processing speed; each test case is also executed 5 times to take the median.

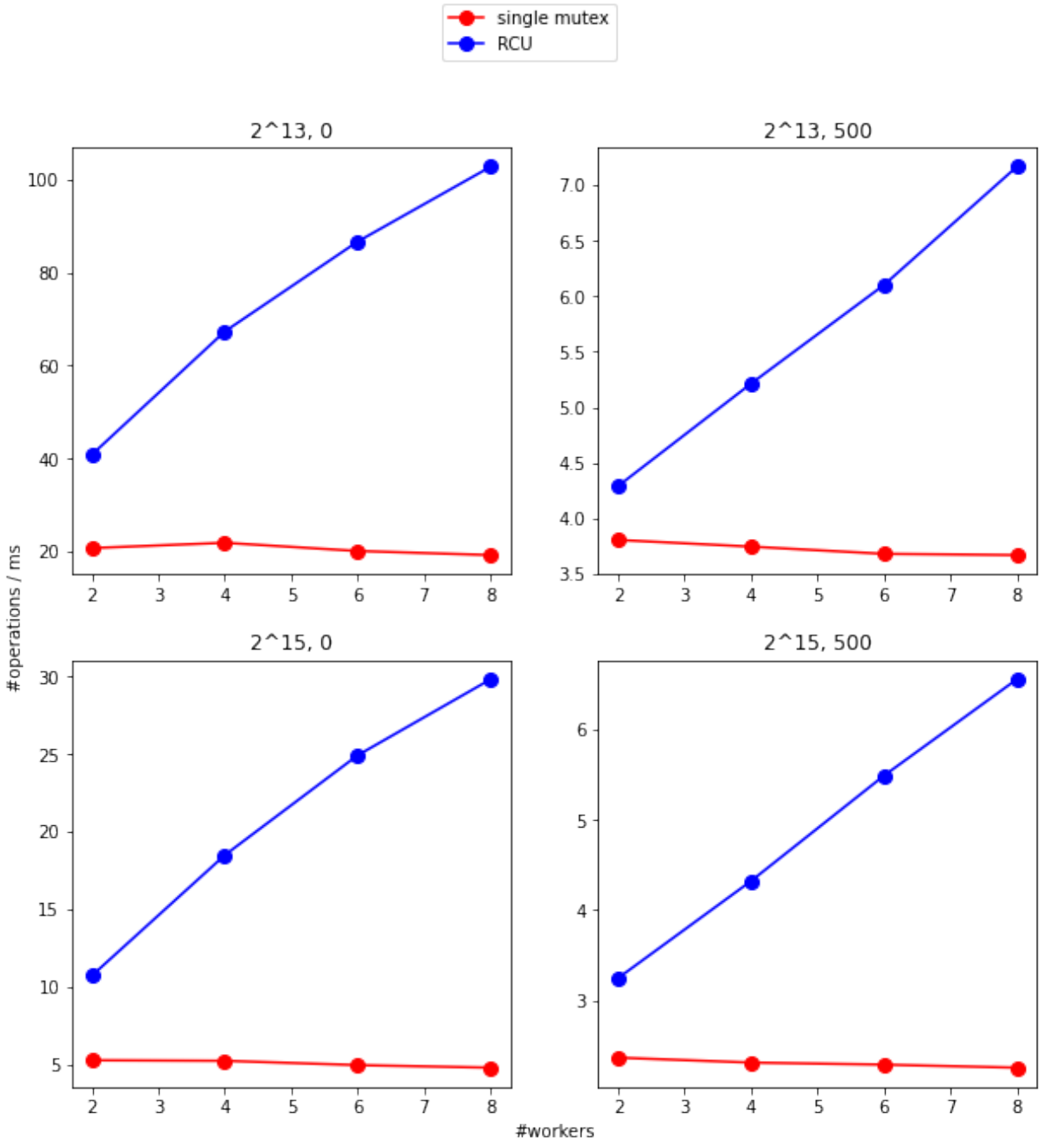

Figure 1: Experiment Result

The graphs represent certain configurations identified by the hash list average length ($2^{13}, 2^{15}$) and the execution time of the per-object operation ($0, 500$ microseconds). The graphs illustrate the correlation between the processing speed and the number of threads ($2, 4, 6, 8$) for each locking strategy.

As expected from the previous analysis, the "single mutex" strategy doesn't gain any extra processing speed despite more threads being given. This implies that each operation's expected processing time will increase linearly with the number of concurrent accesses. In contrast, the RCU hash table manifests approximately linear growth. In other words, each operation's expected processing time is not affected by the number of parallel accesses. Therefore, the RCU hash table is much more suitable for usages where the demand of parallel operations is high.

3.0 Conclusions

The RCU synchronization mechanism allows concurrency between a single writer and multiple readers. By separating the writer into the update phase, the Grace Period, and the reclamation phase, this design reduces the interconnection between the writer and the readers; thus allowing extremely low overhead readers. Furthermore, the `call_rcu` API also allows registration of the reclamation phase in a separate dedicated thread, so that the writers can return before the Grace Period.

The proposed RCU hash table separates the synchronization into 1) a per-object spinlock for exclusive access to an object; 2) the RCU for the hash lists. The RCU hash table exhibits excellent concurrent performance: the number of parallel accesses has almost zero effect on per-thread performance. Therefore, the RCU hash table scales well in demand of increasing parallel operations and demonstrates advantages over the naive "single mutex" hash table.

References

Arcangeli, A., Cao, M., McKenney, P. E., & Sarma, D. (2003). *Using read-copy-update techniques for system v ipc in the linux 2.5 kernel*.

Linux source code. (2021). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v5.15-rc6

McKenney, P. E. (n.d.). *What is RCU, really?* http://www2.rdrop.com/users/paulmck/RCU/whatisRCU.html

McKenney, P. E. (2007). Reply in a discussion. https://lwn.net/Articles/263285/

McKenney, P. E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., & Russell, R. (2002). *Read copy update*. http://www.rdrop.com/users/paulmck/rclock/rcu.2002.07.08.pdf

McKenney, P. E., & Walpole, J. (2007). *What is RCU, fundamentally?* http://lwn.net/Articles/262464/

What is rcu? – read, copy, update. (n.d.). https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

Acknowledgements

I would like to thank my supervisor David Sarrazin for his guidance on implementing the performance comparison. His support is crucial to the experiment and the analysis of the report.