# Read-Copy Update and Its Use in Linux Kernel

*Zijiao Liu*
*Jianxing Chen*
*Zhiqiang Shen*

## Abstract

Our group work is focusing on the classic RCU fundamental mechanisms, usage and API. Evaluation of classic RCU and pre-emptible RCU will also be introduced.

# Contents

# 1. Overview of RCU

RCU, as an efficient synchronization mechanism, was introduced to Linux kernel from version 2.6. RCU supports concurrency between multiple readers and one single writer. In general, readers can read the RCU protected data without lock, but writers should update on a copy version of it; a callback mechanism is used to modify the pointer of the original data to the updated data at an appropriate time. So the read-side almost has zero overhead while RCU updaters can be expensive.

# 2. Fundamental Mechanisms of Classic RCU

The earliest RCU is classic RCU; its mechanism is very simple. There are three fundamental mechanisms of it.

## 2.1 Publish-Subscribe Mechanism

The key feature of RCU is that scanning the protected data safely even the data is being modified. To support the concurrent insertion, RCU adopts Publish-Subscribe Mechanism. Both the writer and reader should have special primitives to implement the mechanism.

The following example (1) well explains the mechanism:

In the writer side:

```
 1 struct foo {
 2 int a;
 3 int b;
 4 int c;
 5 };
 6 struct foo *gp = NULL;
 7
 8 /* . . . */
 9
10 p = kmalloc(sizeof(*p), GFP_KERNEL);
11 p->a = 1;
12 p->b = 2;
13 p->c = 3;
14 gp = p;
```

Consider the pointer gp which is initiated with NULL and a newly allocated and initiated pointer p. gp is to be modified to p in Line 14. Unfortunately, nothing is forcing the CPU and

compiler to execute the operations in the last 4 lines in the writing order. So a RCU primitive rcu_assign_pointer(), which encapsulates memory barrier semantics, is adopted, so the assignment in Line 14 becomes:

```
4 rcu_assign_pointer(gp, p);
```

rcu_assign_pointer() will publish the new data structure and force CPU and compiler to fetch the p after its initialization.

In the reader side:

```
1 p = gp;
2 if (p != NULL) {
3 do_something_with(p->a, p->b, p->c);
4 }
```

It seems to well immune misordering operations by CPU and compiler. Unfortunately, DEC Alpha CPU and value-speculation compiler may fetch the values of p's fields before fetching the value of p! The value-speculation compiler would guess the values of p and fetch the fields of p, then fetch p's real value in order to check if its guess is correct. So rcu_dereference() uses whatever memory barrier and compiler derivatives to server for preventing this kind of misordering:

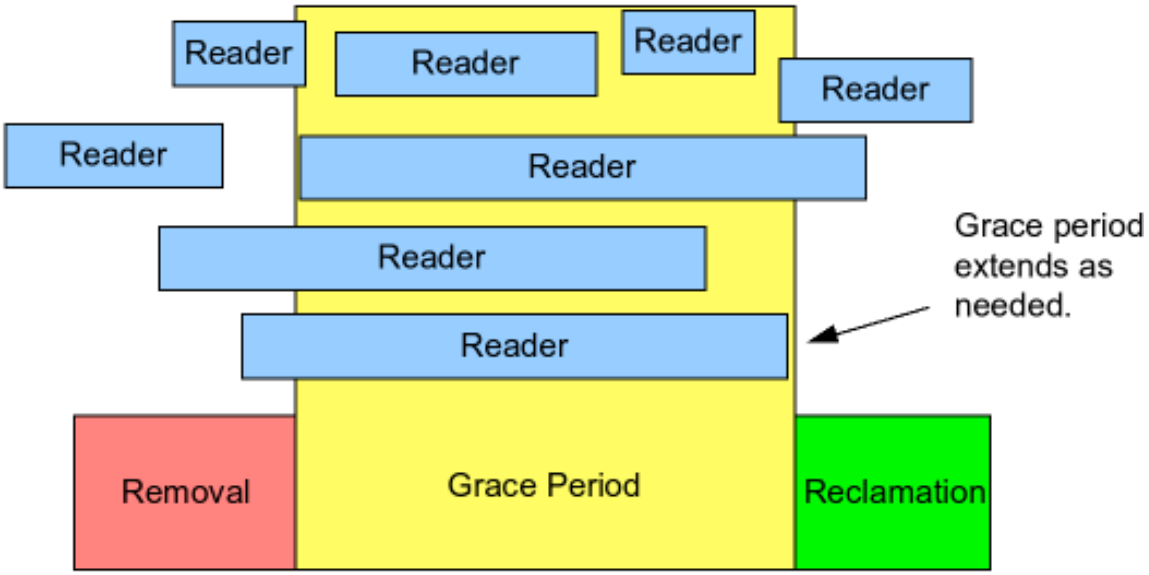
```
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 if (p != NULL) {
4 do_something_with(p->a, p->b, p->c);
5 }
6 rcu_read_unlock();
```

rcu_dereference() can be thought as subscriping the publication of value of the given pointer gp, and ensures that all the subsequent operation can see any initialization gp. rcu_read_lock() and rcu_read_unlock() define the extend of RCU read-side critical section. But they don't spin or block; in fact, they don't generate any code in the non-preemptible kernel, this would be introduced later.

## 2.2 Waiting for All Pre-existing RCU Readers to Complete

RCU maintains multiple versions of data structure in place to accommodate pre-existing readers, and reclaims them after all pre-existing readers complete. A time period during which all such pre-existing readers complete is called a "grace period" (2).

The following graph depicts the grace period of RCU (1).




There are four readers reading on an old version of a RCU protected data, which is removed but not reclaimed. So the grace period must extend to the point that the last readers finish its reading on it. The readers, which begin their reading at the time after the grace period begins, will not see the old version of the removed object; they have nothing with the grace period.

There is a trick on the classic RCU to determine when to really reclaim the old versions of the data structure. Since it is not allowed to explicitly block or sleep in the classic RCU, RCU determine the finish of pre-existing readers by CPU context switch, which means, if each CPU executes at least one context switch, it is guaranteed that all the pre-existing critical sections are quitted, and then the old version can be reclaimed safely.

### 2.3 Maintain Multiple Version of Recently Updated Objects

Readers may see different versions of a RCU protected data structure when reads and updates occur concurrently, this is depending on the time of the readers begin their critical section. Examples are showed in our presentation, so it won't be repeated here.

## 3. Characteristics and Usage of RCU

Key points need to know:

1) In this paper, we are mainly discussing the implementation and characteristic of RCU under the non-CONFIG_PREEMPT Linux kernel. There are several approaches of Linux kernel (3), but when we use RCU primitives, the primitives would automatically disable the

preemptive property of the kernel. Hence, we focus on the applications on the non-preemptive version.

2) In the environment of non-CONFIG_PREEMPT Linux kernel, the implementation of RCU read-side primitives never spins or blocks (Figure 1). That is, the rcu_read_lock and rcu_read_unlock actually do nothing, which provide extremely good overheads. Only synchronize_rcu produce some light weight overhead when it waits for context switch of processors.

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }
void call_rcu(void (*callback) (void *), void *arg)
{
        // add callback/arg pair to a list
}
void synchronize_rcu(void)
{
         int cpu, ncpus = 0;
         for_each_cpu(cpu)
                 schedule_current_task_to(cpu);
         for each entry in the call_rcu list
                 entry->callback (entry->arg);
}
```

**Figure 1 The Implementation of RCU Read-Side Primitives**

In section 3.1, we will see the characteristic of RCU by comparing it to the traditional read-write lock. In section 3.2, we will learn how RCU can act as reference counting mechanism. In section 3.3, we will see some issues of RCU in different Linux kernel and would see some basic solutions regarding these issues.

## 3.1 RCU Act as a Replacement of Traditional Read-Write Lock

Since RCU is a locking mechanism, and it also permits RCU readers to accomplish useful work when running concurrently with RCU updaters, it makes the performance of RCU similar to the traditional read-write lock. So how RCU performs comparing to the traditional read-write lock? Here we will discuss this in the following three aspects: the performance, the deadlock immunity property and real-time latency of RCU.

### *Performance*

From Key Point 1), we can know that RCU can have extremely good reading performance when we do frequent request to the read side primitive. From Part III of this paper, we could get the conclusion from the result that performances of RCU read-side primitives overwhelm traditional read lock. We can see more analysis of this in Part 6. However, RCU does not

have any write-side primitive which looks traditional write lock. That is, their update-side primitives need other locks to keep threads safe. They don't have any advantage when they do update. Hence, it limits RCU can only replace read-write lock in the read intensive scenario.

### *Deadlock Immunity*

From another important attribute of Key Point 2), we can infer this importance characteristic of RCU. Because of the fact that RCU read-side primitive never spin, block, or even do backward branches, the execution time is deterministic. And hence there is no way for a read-side deadlock happened.

We can see a sample code to have a better understanding of this property.

```
 1 rcu_read_lock();
 2 list_for_each_entry_rcu(p, head, list_field) {
 3   do_something_with(p);
 4   if (need_update(p)) {
 5     spin_lock(&my_lock);
 6     do_update(p);
 7     spin_unlock(&my_lock);
 8   }
 9 }
10 rcu_read_unlock();
```

**Figure 2 RCU Read-side Critical Section without Deadlock**

Line 6 is doing whatever relative to pointer/object p. If we implement this code using traditional read-write lock instead, we are definitely in a deadlock situation because we would embed write-lock inside the read-side critical section. In this RCU version code, however, rcu_read_lock at line 1, and rcu_read_unlock at line 10 never spin or block. Also due to the RCU allows other readers to keep their version of data until grace period elapse, the updater does not need to wait the critical section ends and just need to do update regularly under the protection of other kind of locks, say spin lock.

Also note that from this deadlock immunity property, we can easily infer another important property, which is priority inversion immunity. We do not block or spin on a program, so we can prevent low-priority reader from blocking a high priority RCU updater to update data. We can also prevent low-priority updater from blocking high priority readers to read old data.

### *Real-time Latency*

Also from Key Point 2), RCU read-side primitives never spin nor block, it is necessary that RCU readers and RCU updaters must run concurrently. In this situation, RCU readers may access the old data and might even see inconsistencies in the program.

In Figure 3, the readers with white color denote that they are reading old data. The ones with green color denote that they are reading new data. We can see that some RCU readers were reading old data while some were already accessing new version.

However, in a large amount of situation, these inconsistencies are not problems. Because these inconsistencies will only last for sever milliseconds, right after the grace period elapse, the new data would be accessed. Also the RCU updater do not have to wait for the finishes of RCU readers, it can start its update as soon as it comes into schedule. These two characteristics boost RCU to see changes more quickly than would batch-fair read-writer locking readers.

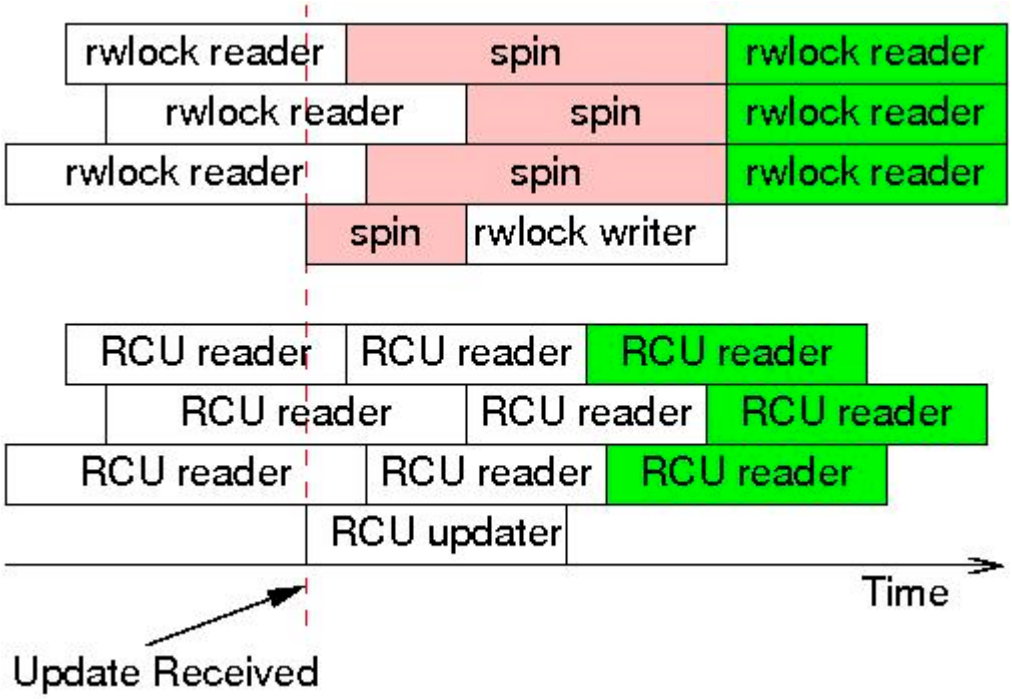



Figure 3 Read-Write Locking Readers and RCU Readers in Real Time

In fact Read-write lock and RCU are just providing different guarantees. Read-write locking readers begin after the writer starts guarantees to see the changes. However, in the read intensive scenario, we are more eager to access data even while the updates are not complete. RCU readers begin after RCU update completes guarantees to see the changes, which would satisfy our demand.

## 3.2 RCU act as a Reference Counting Mechanism

Because the grace period cannot elapse until all previous RCU readers exit their critical sections, it offers the possibility that RCU can be used in reference counting (4).

Reference counting is trying to count how many references are pointing to an object. If the counter reaches zero, this object would be reclaimed. When we want to use a counter in the concurrency scenario, we must use locks to protect the counter, and the lock will degrade the performance of the program. So here we can use RCU as a replacement.

```
1 rcu_read_lock();  /* acquire reference. */
2 p = rcu_dereference(head);
3 /* do something with p. */
4 rcu_read_unlock();  /* release reference. */
```

Figure 4 the Code to "count" the references

```
1 spin_lock(&mylock);
2 p = head;
3 head = NULL;
4 spin_unlock(&mylock);
5 synchronize_rcu();  /* Wait for all references to be
released. */
6 kfree(p);
```

Figure 5 the Code to reclaim the object

When we reference an object, instead of using a counter, we use rcu_read_lock primitive to note as we are acquiring the reference (line 1 in Figure 4), and after everything is done, we use rcu_read_unlock to note as we are releasing the reference (line 4 in Figure 4). By using RCU read-side primitives to protect the object, we are providing existence guarantee (5) for the object. Because we always keep the copy of the data element until the grace period elapses, this data element is guaranteed to remain in existence for the duration of that RCU read-side critical section. Hence, rcu_read_unlock primitive is meaningful to decrease the critical sections which may hold that object. In this case, what we need to do just make sure that all references are released before the object delete. We can use synchronize_rcu primitive (line 5 in Figure 5) to realize this function. After that, the object can be deleted safely (line 6 in Figure 5).

## 4. Approaches of RCU in different Linux Kernel

Up to this point, we are discussing the mechanism and characteristics based on the Clasic RCU. However, there are some issues when we want to apply it into other Linux kernel, say, the CONFIG_PREEMPT kernel. Here we just discuss some simple issues and approaches which would use frequently in reality.

## 4.1 Preemptible RCU

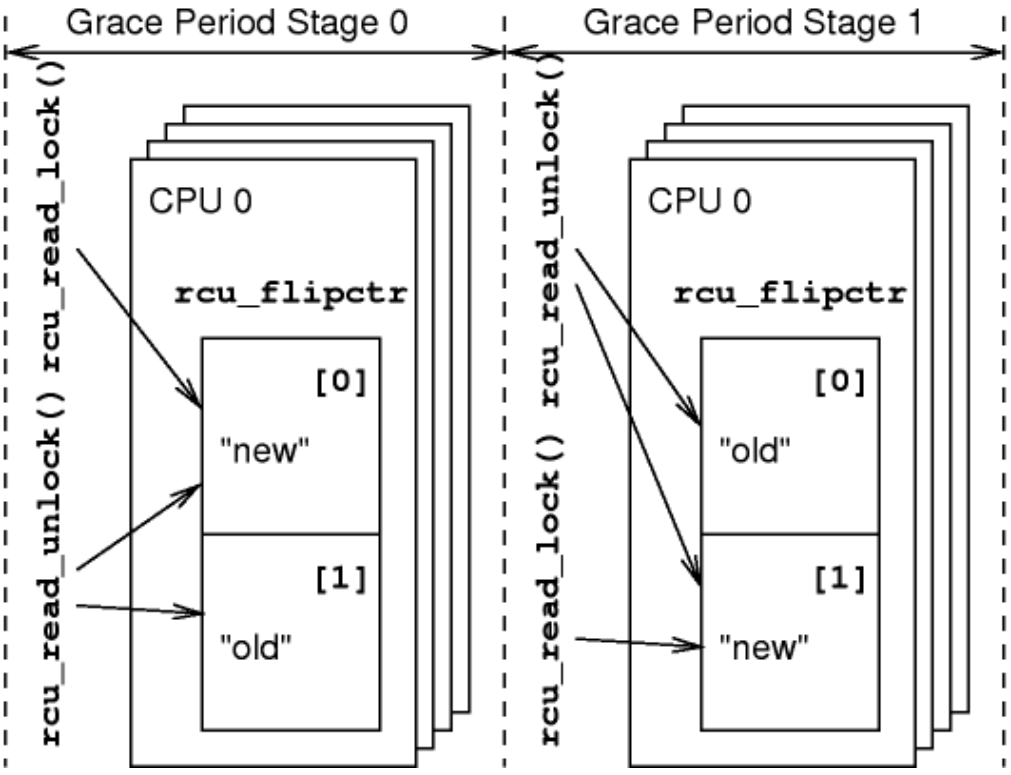
We still begin the discussion with the classic RCU. In the classic RCU mechanism, rcu_read_lock() doesn't really lock the protected data structure, it in fact prohibits preemption. Prohibition of preemption on a CPU is equivalent to prohibit its context switch. A thread, which is running on one preemption prohibited CPU, won't stop, unless the preemption is permitted again or normally quits. rcu_read_unlock() recovers preemption, so CPU can do a context switch and execute update callback.

The mechanism of classic RCU is simple but has downsides: Prohibiting of the preemption by readers degrades system interactivity, and of course degrades the system performance. So preemptible RCU is designed to resolve the problems. Preemptible RCU is design as a self-managed module which can protect the data structure by itself without other system mechanisms, and in preemptible case, it is not necessary to prohibit preemption in the read-side critical section; New data structure and logic control mechanism is introduced; Grace period is divided in to two stages which are implemented by counters.

### *Two Stages Grace Period*

In preemptible RCU, a full grace period consists of two stages.rcu_flipctr, an array with two elements and each one is a counter, is used to determine when a grace period stage ends.

The flowing graph shows the two stages (2):



In grace-period stage 0, rcu_flipctr[0] tracks the new RCU read-side critical sections, namely those starting during the current grace-period stage, and its incremented by rcu_read_lock() and decremented by rcu_read_unlock(); rcu_flipctr[3] tracks old RCU read-side critical sections which started before the current grace-period stage, it is only decrements by rcu_read_unlock() and never incremented at all.

At some time point, some CPU's old rce_flipctr[3] may remain non-zero or even negative, but the sum of all CPU's old rcu_flipctr[3] must eventually go to zero. For example, a task calls rcu_read_lock() on one CPU, but it is preempted during its execution; later, it resumes on another CPU and then calls rcu_read_unlock(). The first CPU's counter will then be +1 and the second CPU's counter will be -1, however, they will still sum to zero. When the sum of the old counter elements does go to zero, it is safe to move to the next grace-period stage no matter what the preemption is.

The array elements switch roles at the beginning of each new grace-period stage. So in the grace-period stage 1, rcu_flipctr[0] tracks the old read-side critical sections, the ones that started in grace-period stage 0; and rcu_flipctr[3] tracks the new read-side critical sections,. The rcu_read_lock() executed during grace-period stage 0, must have a matching rcu_read_unlock() in the grace-period stage 1 if it counted by the counter for old.

The critical point is that all rcu_flipctr elements tracking the old RCU read-side critical sections must strictly decrease. Therefore, once the sum of these old counters reaches zero, it can change.

So, the preemptible RCU emphasize the two grace-period stages instead of grace period. The great advantage of preemptible RCU is that it decouples with the system preemption and improves system performance.

## 4.2 Priority boosting

As mentioned earlier in this paper, RCU has the property of priority inversion immunity in the non-CONFIG_PREEMPT kernel. However, it is susceptible to more subtle priority-inversion scenarios in CONFIG_PREEMPT –rt kernel. In this kernel, the RCU implementation permits read-side critical sections to be blocked waiting for locks or due to preemption. If these critical sections are blocked for too long, grace periods will be stalled, and the amount of memory awaiting the end of a grace period will continually increase, eventually resulting in an out-of-memory condition.

Approach to solve this problem is using RCU Priority Boosting. The full version you can see reference (6). The main idea can be concluded as following: There is a global rcu_boost_prio variable that keeps track of what the RCU readers that schedule out should boost their

priority to. Along with that is every CPU would have a rcu_boost_dat structure to keep its own copy of rcu_boost_prio.

When a RCU reader is preempted and then schedules, it adds itself to a “toboost” list of the local CPU. Let’s say reader R1 is about to schedule and reader R2 is about to sleep. If rcu_boost_prio of R1 is lower than the one of R2, R1 will increase its priority to that of R2. R1 will keep this priority until its critical section ends.

Now when synchronize_rcu is called, if the global rcu_boost_prio is higher than the caller’s priority, caller will update its priority to rcu_boost_prio. And an interation is done to all CPUs rcu_boost_dat. If there are any RCU readers already on the “boosted” list, they would be moved back to “toboost” list. When the priority is set to new rcu_boost_prio, it would be moved to “boosted” list. The Priority Boost is done until no more tasks on the “toboost” list.

After Priority Boost, all the current readers are in same priority and hence the program could run normally.

# 5. The RCU API

The implementation of RCU in Linux 2.6 kernel is among the better-known RCU implementations. As discuss in previous of this paper, the core primitives of RCU is quite small, only five core APIs. All the other APIs can be expressed in terms of these five core APIs.

## 5.1 Core API

The five core RCU APIs are explained briefly below.

rcu_read_lock(): Used by reader to inform the reclaimer that the reader is entering an read-side critical section. Marks an RCU-protected data structure so that it won't be reclaimed for the full duration of read-side critical section.

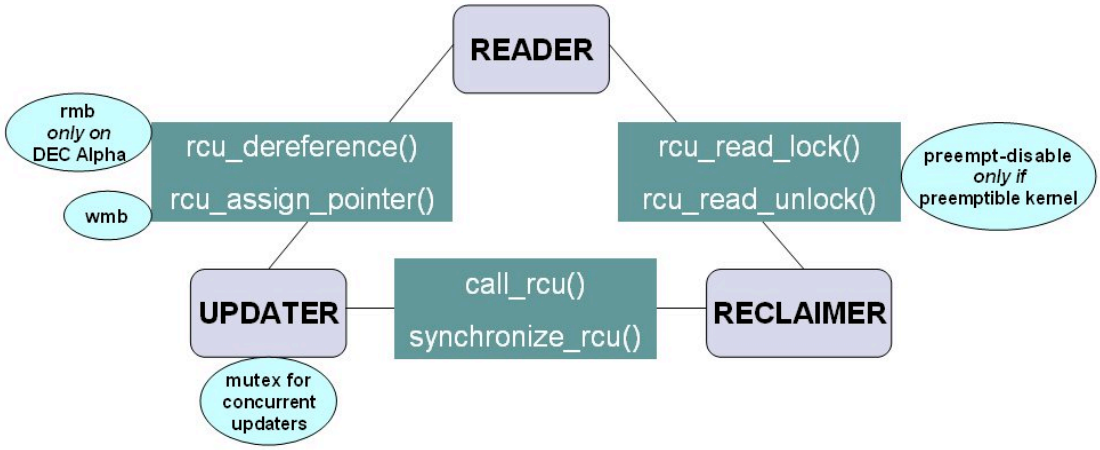
rcu_read_unlock(): Used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.

synchronize_rcu(): Marks the end of updater code and the start of reclaimer code. It blocks until all pre-existing RCU read-side critical sections have completed. Note that synchronize_rcu will not necessarily wait for any subsequent RCU read-side critical sections to complete.

rcu_assign_pointer(): The updater uses this function to assign a new value to an RCU-protected pointer, in order to safely communicate the change in value from the updater to the reader. This function returns the new value, and also executes any memory barrier instructions required for a given CPU architecture.

rcu_dereference_pointer(): The reader uses rcu_dereference_pointer to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. It also executes any needed memory-barrier instructions for a given CPU architecture. The value returned by rcu_dereference_pointer is valid only within the enclosing RCU read-side critical section. As with rcu_assign_pointer, an important function of rcu_dereference_pointer is to document which pointers are protected by RCU.

The following figure (7) shows how each of core RCU API interactive among the reader, updater and reclaimer.




## 5.2 Full family of API

### 5.2.1 Wait-to-Finish APIs

The most straightforward answer to “what is RCU” is that RCU is an API used in the Linux kernel, as summarized by the Table 1 and the following discussion. Or, more precisely, RCU is a four-member family of APIs as shown in the table, with each column corresponding to one of the family members.

| Attribute | RCU | RCU BH | RCU Sched | SRCU |
|---|---|---|---|---|
| **Purpose** | Wait for RCU read-side critical sections | Wait for RCU-bh read-side critical sections & irqs | Wait for RCU-sched read-side critical sections, preempt-disable regions, hardirqs, & NMIs | Wait for SRCU read-side critical sections, allow sleeping readers |
| **Read-side primitives** | rcu_read_lock() rcu_read_unlock() | rcu_read_lock_bh() rcu_read_unlock_ | rcu_read_lock_sched() rcu_read_unlock_ | srcu_read_lock() srcu_read_unlo |

| | | | | |
|---|---|---|---|---|
| | | bh() | sched()<br>rcu_read_lock_sched_notrace()<br>rcu_read_unlock_sched_notrace()<br>preempt_disable()<br>preempt_enable()<br>(and friends) | ck() |
| **Update-side primitives (synchronous)** | synchronize_rcu()<br>synchronize_net() | synchronize_rcu_bh() | synchronize_sched() | synchronize_srcu() |
| **Update-side primitives (expedited)** | synchronize_rcu_expedited() | synchronize_rcu_bh_expedited() | synchronize_sched_expedited() | synchronize_srcu_expedited() |
| **Update-side primitives (asynchronous/callback)** | call_rcu() | call_rcu_bh() | call_rcu_sched() | N/A |
| **Update-side primitives (wait for callbacks)** | rcu_barrier() | rcu_barrier_bh() | rcu_barrier_sched() | N/A |
| **Read side constraints** | No blocking except preemption and “spinlock” acquisition. | No BH enabling | No blocking | No wait for synchronize_srcu() |
| **Read side overhead** | Simple instructions (free on !PREEMPT) | BH disable/enable | Preempt disable/enable (free on !PREEMPT) | Simple instructions, preempt disable/enable |
| **Asynchronous update-side overhead (for example, call_rcu())** | sub-microsecond | sub-microsecond | sub-microsecond | N/A |
| **Grace-period latency** | 10s of milliseconds | 10s of milliseconds | 10s of milliseconds | 10s of milliseconds |
| **!PREEMPT_RT default implementation** | RCU Sched | RCU BH | RCU Sched | SRCU |
| **PREEMPT_RT default implementation** | Preemptible RCU | RCU BH | RCU Sched | SRCU |

Table 1 The RCU API table

### 5.2.2 Publish-Subscribe and Version-Maintenance APIs

The RCU publish-subscribe and version-maintenance primitives shown in the following Table 2 apply to all of the variants of RCU discussed above.

| Category | Primitives | Availability | Overhead |
|---|---|---|---|
| **List traversal** | list_for_each_entry_rcu() | 2.5.59 | Simple instructions (memory barrier on Alpha) |
| **List update** | list_add_rcu() | 2.5.44 | Memory barrier |
| | list_add_tail_rcu() | 2.5.44 | Memory barrier |
| | list_del_rcu() | 2.5.44 | Simple instructions |
| | list_replace_rcu() | 2.6.9 | Memory barrier |
| | list_splice_init_rcu() | 2.6.21 | Grace-period latency |
| **Hlist traversal** | hlist_for_each_entry_rcu() | 2.6.8 | Simple instructions (memory barrier on Alpha) |
| **Hlist update** | hlist_add_after_rcu() | 2.6.14 | Memory barrier |
| | hlist_add_before_rcu() | 2.6.14 | Memory barrier |
| | hlist_add_head_rcu() | 2.5.64 | Memory barrier |
| | hlist_del_rcu() | 2.5.64 | Simple instructions |
| | hlist_replace_rcu() | 2.6.15 | Memory barrier |
| **Pointer traversal** | rcu_dereference() | 2.6.9 | Simple instructions (memory barrier on Alpha) |
| **Pointer update** | rcu_assign_pointer() | 2.6.10 | Memory barrier |

Table 2 Publish-Subscribe and Version-Maintenance APIs

These APIs are mostly used in kernel application; provide a means to access list data structure. Read-mostly list-based data structures that can tolerate stale data are the most amenable to use of RCU. The most celebrated example is the routing table. Because the routing table is tracking the state of equipment outside of the computer, it will at times contain stale data.

## 5.3 Analogy with Reader-Writer Locking

One of the best applications of RCU is to protect read-mostly linked lists. The most common use of RCU is to replace the reader-writer locking. And we will see a straightforward example of this use of RCU in real Linux kernel code. The following codes are reader-writer locked and RCU implementation of audit_filter_task() function, may be found in the system-call auditing support.

```
struct audit_entry *e;
enum audit_state   state;
read_lock(&auditsc_lock);
list_for_each_entry(e, &audit_tsklist, list) {
  if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
    read_unlock(&auditsc_lock);
    return state;
```

```
  }
}
read_unlock(&auditsc_lock);
return AUDIT_BUILD_CONTEXT;
```

Figure 6 a reader-writer locked implementation of audit_filter_task()

```
struct audit_entry *e;
enum audit_state   state;
rcu_read_lock();
list_for_each_entry_rcu(e, &audit_tsklist, list) {
  if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
    rcu_read_unlock();
    return state;
  }
}
rcu_read_unlock();
return AUDIT_BUILD_CONTEXT;
```

Figure 7 a RCU implementation of audit_filter_task()

From above codes, the read_lock() and read_unlock() calls have replaced by rcu_read_lock() and rcu_read_unlock(), respectively, and the list_for_each_entry() has become list_for_each_entry_rcu(). Either way, the differences are quite small.

## 6. Benchmark

Since RCU is analogous to reader-writer locking, we will benchmark these two implementations, reader-writer lock and RCU. The following is the actual code to do the benchmark. Please note that we only show parts of the code, the read-side code, here to illustrate the idea behind these two methods.

```
for (i = 0; i < OUTER_READ_LOOP; i++) {
    for (j = 0; j < INNER_READ_LOOP; j++) {
      pthread_rwlock_rdlock(&lock);
      assert(test_array.a == 8);
      pthread_rwlock_unlock(&lock);
    }
  }
```

Figure 8 Read-side of Reader-Writer Lock

```
for (i = 0; i < OUTER_READ_LOOP; i++) {
    for (j = 0; j < INNER_READ_LOOP; j++) {
      rcu_read_lock();
      local_ptr = rcu_dereference(test_rcu_pointer);
      if (local_ptr) {
```

```
        assert(local_ptr->a == 8);
      }
      rcu_read_unlock();
    }
  }
```

Figure 9 Read-side of RCU Lock

The benchmarks were run on the energon3.cims.nyu.edu machine. And each reader thread iterates 200000000 times of read, and each writer thread iterates 2000 times of write. We run 10 reader threads and 10 writer threads for each benchmark. The result shows in the following table. We can see that the read performance of RCU increased significantly over Reader-Writer Lock.

| | Reader-Writer Lock | RCU Lock |
|---|---|---|
| **Time per read** | 7150.19 cycles | 86.586 cycles |
| **Time per write** | 6.11023e+07 cycles | 1.00858e+07 cycles |

Table 3 Benchmark result

## Works Cited

1. **McKenney, Paul E.** What is RCU, Fundamentally. [Online] http://lwn.net/Articles/262464/.

2. —. The design of preemptible read-copy-update. [Online] http://lwn.net/Articles/253651/.

3. —. Seven real-time Linux approaches. [Online] http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Seven-realtime-Linux-approaches-Part-C/.

4. **daper.** C++ object's reference counting. [Online] http://www.linuxprogrammingblog.com/cpp-objects-reference-counting.

5. **Ben Gamsa, Orran Krieger, Jonathan Appavoo, Micheal Stumm.** Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation.* February 1999.

6. **McKenney, Paul E.** Priority-Boosting RCU Read-Side Critical Sections. [Online] http://lwn.net/Articles/220677/.

7. **Wikipedia.** Read-copy-update. [Online] http://en.wikipedia.org/wiki/Read-copy-update.