# Using RCU to Protect Read-Mostly Linked Lists

One of the best applications of RCU is to protect read-mostly linked lists (`struct list_head` in list.h). One big advantage of this approach is that all of the required memory barriers are included for you in the list macros. This document describes several applications of RCU, with the best fits first.

## Example 1: Read-mostly list: Deferred Destruction

A widely used usecase for RCU lists in the kernel is lockless iteration over all processes in the system. `task_struct::tasks` represents the list node that links all the processes. The list can be traversed in parallel to any list additions or removals.

The traversal of the list is done using `for_each_process()` which is defined by the 2 macros:

```
#define next_task(p) \
        list_entry_rcu((p)->tasks.next, struct task_struct, tasks)

#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

The code traversing the list of all processes typically looks like:

```
rcu_read_lock();
for_each_process(p) {
        /* Do something with p */
}
rcu_read_unlock();
```

The simplified code for removing a process from a task list is:

```
void release_task(struct task_struct *p)
{
        write_lock(&tasklist_lock);
        list_del_rcu(&p->tasks);
        write_unlock(&tasklist_lock);
        call_rcu(&p->rcu, delayed_put_task_struct);
}
```

When a process exits, `release_task()` calls `list_del_rcu(&p->tasks)` under `tasklist_lock` writer lock protection, to remove the task from the list of all tasks. The `tasklist_lock` prevents concurrent list additions/removals from corrupting the list. Readers using `for_each_process()` are not protected with the `tasklist_lock`. To prevent readers from noticing changes in the list pointers, the `task_struct` object is freed only after one or more grace periods elapse (with the help of call_rcu()). This deferring of destruction ensures that any readers traversing the list will see valid `p->tasks.next` pointers and deletion/freeing can happen in parallel with traversal of the list. This pattern is also called an **existence lock**, since RCU pins the object in memory until all existing readers finish.

## Example 2: Read-Side Action Taken Outside of Lock: No In-Place Updates

The best applications are cases where, if reader-writer locking were used, the read-side lock would be dropped before taking any action based on the results of the search. 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. Therefore, once the route has been computed, there is no need to hold the routing table static during transmission of the packet. After all, you can hold the routing table static all you want, but that won't keep the external Internet from changing, and it is the state of the external Internet that really matters. In addition, routing entries are typically added or deleted, rather than being modified in place.

A straightforward example of this use of RCU may be found in the system-call auditing support. For example, a reader-writer locked implementation of `audit_filter_task()` might be as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        struct audit_entry *e;
        enum audit_state   state;

        read_lock(&auditsc_lock);
        /* Note: audit_filter_mutex held by caller. */
        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;
}
```

Here the list is searched under the lock, but the lock is dropped before the corresponding value is returned. By the time that this value is acted on, the list may well have been modified. This makes sense, since if you are turning auditing off, it is OK to audit a few extra system calls.

This means that RCU can be easily applied to the read side, as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        struct audit_entry *e;
        enum audit_state   state;

        rcu_read_lock();
        /* Note: audit_filter_mutex held by caller. */
        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;
}
```

The `read_lock()` and `read_unlock()` calls have become rcu_read_lock() and rcu_read_unlock(), respectively, and the list_for_each_entry() has become list_for_each_entry_rcu(). The **_rcu()** list-traversal primitives insert the read-side memory barriers that are required on DEC Alpha CPUs.

The changes to the update side are also straightforward. A reader-writer lock might be used as follows for deletion and insertion:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        write_lock(&auditsc_lock);
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del(&e->list);
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
                                 struct list_head *list)
{
        write_lock(&auditsc_lock);
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add(&entry->list, list);
        } else {
                list_add_tail(&entry->list, list);
        }
        write_unlock(&auditsc_lock);
        return 0;
}
```

Following are the RCU equivalents for these two functions:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        /* No need to use the _rcu iterator here, since this is the only
         * deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del_rcu(&e->list);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
                                 struct list_head *list)
```

```
{
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add_rcu(&entry->list, list);
        } else {
                list_add_tail_rcu(&entry->list, list);
        }
        return 0;
}
```

Normally, the `write_lock()` and `write_unlock()` would be replaced by a spin_lock() and a spin_unlock(). But in this case, all callers hold `audit_filter_mutex`, so no additional locking is required. The `auditsc_lock` can therefore be eliminated, since use of RCU eliminates the need for writers to exclude readers.

The list_del(), list_add(), and list_add_tail() primitives have been replaced by list_del_rcu(), list_add_rcu(), and list_add_tail_rcu(). The **_rcu()** list-manipulation primitives add memory barriers that are needed on weakly ordered CPUs (most of them!). The list_del_rcu() primitive omits the pointer poisoning debug-assist code that would otherwise cause concurrent readers to fail spectacularly.

So, when readers can tolerate stale data and when entries are either added or deleted, without in-place modification, it is very easy to use RCU!

## Example 3: Handling In-Place Updates

The system-call auditing code does not update auditing rules in place. However, if it did, the reader-writer-locked code to do so might look as follows (assuming only `field_count` is updated, otherwise, the added fields would need to be filled in):

```
static inline int audit_upd_rule(struct audit_rule *rule,
                                 struct list_head *list,
                                 __u32 newaction,
                                 __u32 newfield_count)
{
        struct audit_entry *e;
        struct audit_entry *ne;

        write_lock(&auditsc_lock);
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        e->rule.action = newaction;
                        e->rule.field_count = newfield_count;
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}
```

The RCU version creates a copy, updates the copy, then replaces the old entry with the newly updated entry. This sequence of actions, allowing concurrent reads while making a copy to perform an update, is what gives RCU (*read-copy update*) its name. The RCU code is as follows:

```
static inline int audit_upd_rule(struct audit_rule *rule,
                                 struct list_head *list,
                                 __u32 newaction,
                                 __u32 newfield_count)
{
        struct audit_entry *e;
        struct audit_entry *ne;

        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        ne = kmalloc(sizeof(*entry), GFP_ATOMIC);
                        if (ne == NULL)
                                return -ENOMEM;
                        audit_copy_rule(&ne->rule, &e->rule);
                        ne->rule.action = newaction;
                        ne->rule.field_count = newfield_count;
                        list_replace_rcu(&e->list, &ne->list);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

Again, this assumes that the caller holds `audit_filter_mutex`. Normally, the writer lock would become a spinlock in this sort of code.

Another use of this pattern can be found in the openswitch driver's *connection tracking table* code in `ct_limit_set()`. The table holds connection tracking entries and has a limit on the maximum entries. There is one such table per-zone and hence one *limit* per zone. The zones are mapped to their limits through a hashtable using an RCU-managed hlist for the hash chains. When a new limit is set, a new limit object is allocated and `ct_limit_set()` is called to replace the old limit object with the new one using list_replace_rcu(). The old limit object is then freed after a grace period using kfree_rcu().

## Example 4: Eliminating Stale Data

The auditing example above tolerates stale data, as do most algorithms that are tracking external state. Because there is a delay from the time the external state changes before Linux becomes aware of the change, additional RCU-induced staleness is generally not a problem.

However, there are many examples where stale data cannot be tolerated. One example in the Linux kernel is the System V IPC (see the shm_lock() function in ipc/shm.c). This code checks a *deleted* flag under a per-entry spinlock, and, if the *deleted* flag is set, pretends that the entry does not exist. For this to be helpful, the search function must return holding the per-entry spinlock, as shm_lock() does in fact do.

Quick Quiz:

For the deleted-flag technique to be helpful, why is it necessary to hold the per-entry lock while returning from the search function?

:ref:`Answer to Quick Quiz <quick_quiz_answer>`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\(linux-master) (Documentation) (RCU)listRCU.rst`, **line 301**); ***backlink***
>
> Unknown interpreted text role "ref".

If the system-call audit module were to ever need to reject stale data, one way to accomplish this would be to add a `deleted` flag and a `lock` spinlock to the audit_entry structure, and modify `audit_filter_task()` as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        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)) {
                        spin_lock(&e->lock);
                        if (e->deleted) {
                                spin_unlock(&e->lock);
                                rcu_read_unlock();
                                return AUDIT_BUILD_CONTEXT;
                        }
                        rcu_read_unlock();
                        return state;
                }
        }
        rcu_read_unlock();
        return AUDIT_BUILD_CONTEXT;
}
```

Note that this example assumes that entries are only added and deleted. Additional mechanism is required to deal correctly with the update-in-place performed by `audit_upd_rule()`. For one thing, `audit_upd_rule()` would need additional memory barriers to ensure that the list_add_rcu() was really executed before the list_del_rcu().

The `audit_del_rule()` function would need to set the `deleted` flag under the spinlock as follows:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        /* No need to use the _rcu iterator here, since this
         * is the only deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        spin_lock(&e->lock);
                        list_del_rcu(&e->list);
                        e->deleted = 1;
                        spin_unlock(&e->lock);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
```

```
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

This too assumes that the caller holds `audit_filter_mutex`.

## Example 5: Skipping Stale Objects

For some usecases, reader performance can be improved by skipping stale objects during read-side list traversal if the object in concern is pending destruction after one or more grace periods. One such example can be found in the timerfd subsystem. When a `CLOCK_REALTIME` clock is reprogrammed - for example due to setting of the system time, then all programmed timerfds that depend on this clock get triggered and processes waiting on them to expire are woken up in advance of their scheduled expiry. To facilitate this, all such timers are added to an RCU-managed `cancel_list` when they are setup in `timerfd_setup_cancel()`:

```
static void timerfd_setup_cancel(struct timerfd_ctx *ctx, int flags)
{
        spin_lock(&ctx->cancel_lock);
        if ((ctx->clockid == CLOCK_REALTIME &&
            (flags & TFD_TIMER_ABSTIME) && (flags & TFD_TIMER_CANCEL_ON_SET)) {
                if (!ctx->might_cancel) {
                        ctx->might_cancel = true;
                        spin_lock(&cancel_lock);
                        list_add_rcu(&ctx->clist, &cancel_list);
                        spin_unlock(&cancel_lock);
                }
        }
        spin_unlock(&ctx->cancel_lock);
}
```

When a timerfd is freed (fd is closed), then the `might_cancel` flag of the timerfd object is cleared, the object removed from the `cancel_list` and destroyed:

```
int timerfd_release(struct inode *inode, struct file *file)
{
        struct timerfd_ctx *ctx = file->private_data;

        spin_lock(&ctx->cancel_lock);
        if (ctx->might_cancel) {
                ctx->might_cancel = false;
                spin_lock(&cancel_lock);
                list_del_rcu(&ctx->clist);
                spin_unlock(&cancel_lock);
        }
        spin_unlock(&ctx->cancel_lock);

        hrtimer_cancel(&ctx->t.tmr);
        kfree_rcu(ctx, rcu);
        return 0;
}
```

If the `CLOCK_REALTIME` clock is set, for example by a time server, the hrtimer framework calls `timerfd_clock_was_set()` which walks the `cancel_list` and wakes up processes waiting on the timerfd. While iterating the `cancel_list`, the `might_cancel` flag is consulted to skip stale objects:

```
void timerfd_clock_was_set(void)
{
        struct timerfd_ctx *ctx;
        unsigned long flags;

        rcu_read_lock();
        list_for_each_entry_rcu(ctx, &cancel_list, clist) {
                if (!ctx->might_cancel)
                        continue;
                spin_lock_irqsave(&ctx->wqh.lock, flags);
                if (ctx->moffs != ktime_mono_to_real(0)) {
                        ctx->moffs = KTIME_MAX;
                        ctx->ticks++;
                        wake_up_locked_poll(&ctx->wqh, EPOLLIN);
                }
                spin_unlock_irqrestore(&ctx->wqh.lock, flags);
        }
        rcu_read_unlock();
}
```

The key point here is, because RCU-traversal of the `cancel_list` happens while objects are being added and removed to the list, sometimes the traversal can step on an object that has been removed from the list. In this example, it is seen that it is better to skip such objects using a flag.

## Summary

Read-mostly list-based data structures that can tolerate stale data are the most amenable to use of RCU. The simplest case is where entries are either added or deleted from the data structure (or atomically modified in place), but non-atomic in-place modifications can be handled by making a copy, updating the copy, then replacing the original with the copy. If stale data cannot be tolerated, then a *deleted* flag may be used in conjunction with a per-entry spinlock in order to allow the search function to reject newly deleted data.

Answer to Quick Quiz:

> For the deleted-flag technique to be helpful, why is it necessary to hold the per-entry lock while returning from the search function?
>
> If the search function drops the per-entry lock before returning, then the caller will be processing stale data in any case. If it is really OK to be processing stale data, then you don't need a *deleted* flag. If processing stale data really is a problem, then you need to hold the per-entry lock across all of the code that uses the value that was returned.

:ref:`Back to Quick Quiz <quick_quiz>`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\(linux-master)(Documentation)(RCU)listRCU.rst`, **line 468**); ***backlink***
>
> Unknown interpreted text role "ref".