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-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_netlink_sem 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_netlink_sem 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;

                /* Do not 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_netlink_sem, 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. Normally, the write_lock() calls would be converted into spin_lock() calls.

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 2: Handling In-Place Updates

The system-call auditing code does not update auditing rules in place. However, if it did, reader-writer-locked code to do so might look as follows (presumably, the field_count is only permitted to decrease, 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_newentry *ne;

		write_lock(&auditsc_lock);
		/* Note: audit_netlink_sem held by caller. */
		list_for_each_entry(e, list, list) {
			if (!audit_compare_rule(rule, &e->rule)) {
				e->rule.action = newaction;
				e->rule.file_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 doing 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_newentry *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.file_count = newfield_count;
				list_replace_rcu(e, ne);
				call_rcu(&e->rcu, audit_free_rule);
				return 0;
			}
		}
		return -EFAULT;		/* No matching rule */
	}
```

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

Example 3: Eliminating Stale Data

The auditing examples above tolerate 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 normally 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 ipc_lock() function in ipc/util.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 ipc_lock() does in fact do.

Quick Quiz: Why does the search function need to return holding the
	per-entry lock for this deleted-flag technique to be helpful?

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;

		/* Do not 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 */
}
```

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

Why does the search function need to return holding the per-entry lock for this deleted-flag technique to be helpful?

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.