Using RCU to Protect Read-Mostly Arrays

Although RCU is more commonly used to protect linked lists, it can also be used to protect arrays. Three situations are as follows:

1. :ref:`Hash Tables <hash_tables>`

   System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\(linux-master)(Documentation)(RCU)arrayRCU.rst, line 9); *backlink*

   Unknown interpreted text role "ref".

2. :ref:`Static Arrays <static_arrays>`

   System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\(linux-master)(Documentation)(RCU)arrayRCU.rst, line 11); *backlink*

   Unknown interpreted text role "ref".

3. :ref:`Resizable Arrays <resizable_arrays>`

   System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\(linux-master)(Documentation)(RCU)arrayRCU.rst, line 13); *backlink*

   Unknown interpreted text role "ref".

Each of these three situations involves an RCU-protected pointer to an array that is separately indexed. It might be tempting to consider use of RCU to instead protect the index into an array, however, this use case is **not** supported. The problem with RCU-protected indexes into arrays is that compilers can play way too many optimization games with integers, which means that the rules governing handling of these indexes are far more trouble than they are worth. If RCU-protected indexes into arrays prove to be particularly valuable (which they have not thus far), explicit cooperation from the compiler will be required to permit them to be safely used.

That aside, each of the three RCU-protected pointer situations are described in the following sections.

Situation 1: Hash Tables

Hash tables are often implemented as an array, where each array entry has a linked-list hash chain. Each hash chain can be protected by RCU as described in the listRCU.txt document. This approach also applies to other array-of-list situations, such as radix trees.

Situation 2: Static Arrays

Static arrays, where the data (rather than a pointer to the data) is located in each array element, and where the array is never resized, have not been used with RCU. Rik van Riel recommends using seqlock in this situation, which would also have minimal read-side overhead as long as updates are rare.

Quick Quiz:
    Why is it so important that updates be rare when using seqlock?

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\(linux-master)(Documentation)(RCU)arrayRCU.rst, line 53); *backlink*

Unknown interpreted text role "ref".

Situation 3: Resizable Arrays

Use of RCU for resizable arrays is demonstrated by the grow_ary() function formerly used by the System V IPC code. The array is used to map from semaphore, message-queue, and shared-memory IDs to the data structure that represents the corresponding IPC construct. The grow_ary() function does not acquire any locks; instead its caller must hold the ids->sem semaphore.

The grow_ary() function, shown below, does some limit checks, allocates a new ipc_id_ary, copies the old to the new portion of the new, initializes the remainder of the new, updates the ids->entries pointer to point to the new array, and invokes ipc_rcu_putref() to

free up the old array. Note that rcu_assign_pointer() is used to update the ids->entries pointer, which includes any memory barriers required on whatever architecture you are running on:

```
static int grow_ary(struct ipc_ids* ids, int newsize)
{
        struct ipc_id_ary* new;
        struct ipc_id_ary* old;
        int i;
        int size = ids->entries->size;

        if(newsize > IPCMNI)
                newsize = IPCMNI;
        if(newsize <= size)
                return newsize;

        new = ipc_rcu_alloc(sizeof(struct kern_ipc_perm *)*newsize +
                            sizeof(struct ipc_id_ary));
        if(new == NULL)
                return size;
        new->size = newsize;
        memcpy(new->p, ids->entries->p,
               sizeof(struct kern_ipc_perm *)*size +
               sizeof(struct ipc_id_ary));
        for(i=size;i<newsize;i++) {
                new->p[i] = NULL;
        }
        old = ids->entries;

        /*
         * Use rcu_assign_pointer() to make sure the memcpyed
         * contents of the new array are visible before the new
         * array becomes visible.
         */
        rcu_assign_pointer(ids->entries, new);

        ipc_rcu_putref(old);
        return newsize;
}
```

The ipc_rcu_putref() function decrements the array's reference count and then, if the reference count has dropped to zero, uses call_rcu() to free the array after a grace period has elapsed.

The array is traversed by the ipc_lock() function. This function indexes into the array under the protection of rcu_read_lock(), using rcu_dereference() to pick up the pointer to the array so that it may later safely be dereferenced -- memory barriers are required on the Alpha CPU. Since the size of the array is stored with the array itself, there can be no array-size mismatches, so a simple check suffices. The pointer to the structure corresponding to the desired IPC object is placed in "out", with NULL indicating a non-existent entry. After acquiring "out->lock", the "out->deleted" flag indicates whether the IPC object is in the process of being deleted, and, if not, the pointer is returned:

```
struct kern_ipc_perm* ipc_lock(struct ipc_ids* ids, int id)
{
        struct kern_ipc_perm* out;
        int lid = id % SEQ_MULTIPLIER;
        struct ipc_id_ary* entries;

        rcu_read_lock();
        entries = rcu_dereference(ids->entries);
        if(lid >= entries->size) {
                rcu_read_unlock();
                return NULL;
        }
        out = entries->p[lid];
        if(out == NULL) {
                rcu_read_unlock();
                return NULL;
        }
        spin_lock(&out->lock);

        /* ipc_rmid() may have already freed the ID while ipc_lock
         * was spinning: here verify that the structure is still valid
         */
        if (out->deleted) {
                spin_unlock(&out->lock);
                rcu_read_unlock();
                return NULL;
        }
        return out;
}
```

Answer to Quick Quiz:

Why is it so important that updates be rare when using seqlock?

The reason that it is important that updates be rare when using seqlock is that frequent updates can livelock readers. One way to avoid this problem is to assign a seqlock for each array entry rather than to the entire array.