Reference-count design for elements of lists/arrays protected by RCU

Please note that the percpu-ref feature is likely your first stop if you need to combine reference counts and RCU. Please see include/linux/percpu-refcount.h for more information. However, in those unusual cases where percpu-ref would consume too much memory, please read on.

Reference counting on elements of lists which are protected by traditional reader/writer spinlocks or semaphores are straightforward:

CODE LISTING A:

```
1.                                      2.
add()                                   search_and_reference()
{                                       {
    alloc_object                            read_lock(&list_lock);
    ...                                     search_for_element
    atomic_set(&el->rc, 1);                 atomic_inc(&el->rc);
    write_lock(&list_lock);                  ...
    add_element                             read_unlock(&list_lock);
    ...                                     ...
    write_unlock(&list_lock);           }
}

3.                                      4.
release_referenced()                    delete()
{                                       {
    ...                                     write_lock(&list_lock);
    if(atomic_dec_and_test(&el->rc))        ...
        kfree(el);
    ...                                     remove_element
}                                           write_unlock(&list_lock);
                                            ...
                                            if (atomic_dec_and_test(&el->rc))
                                                kfree(el);
                                            ...
                                        }
```

If this list/array is made lock free using RCU as in changing the write_lock() in add() and delete() to spin_lock() and changing read_lock() in search_and_reference() to rcu_read_lock(), the atomic_inc() in search_and_reference() could potentially hold reference to an element which has already been deleted from the list/array. Use atomic_inc_not_zero() in this scenario as follows:

CODE LISTING B:

```
1.                                      2.
add()                                   search_and_reference()
{                                       {
    alloc_object                            rcu_read_lock();
    ...                                     search_for_element
    atomic_set(&el->rc, 1);                 if (!atomic_inc_not_zero(&el->rc)) {
    spin_lock(&list_lock);                      rcu_read_unlock();
                                                return FAIL;
    add_element                             }
    ...                                     ...
    spin_unlock(&list_lock);                rcu_read_unlock();
}                                       }
3.                                      4.
release_referenced()                    delete()
{                                       {
    ...                                     spin_lock(&list_lock);
    if (atomic_dec_and_test(&el->rc))       ...
        call_rcu(&el->head, el_free);       remove_element
    ...                                     spin_unlock(&list_lock);
}                                           ...
                                            if (atomic_dec_and_test(&el->rc))
                                                call_rcu(&el->head, el_free);
                                            ...
                                        }
```

Sometimes, a reference to the element needs to be obtained in the update (write) stream. In such cases, atomic_inc_not_zero() might be overkill, since we hold the update-side spinlock. One might instead use atomic_inc() in such cases.

It is not always convenient to deal with "FAIL" in the search_and_reference() code path. In such cases, the atomic_dec_and_test() may be moved from delete() to el_free() as follows:

CODE LISTING C:

```
1.
add()
{
    alloc_object
    ...
    atomic_set(&el->rc, 1);
    spin_lock(&list_lock);

    add_element
    ...
    spin_unlock(&list_lock);
}
```

```
2.
search_and_reference()
{
    rcu_read_lock();
    search_for_element
    atomic_inc(&el->rc);
    ...

    rcu_read_unlock();
}
```

```
3.
release_referenced()
{
    ...
    if (atomic_dec_and_test(&el->rc))
        kfree(el);
    ...
}
```

```
4.
delete()
{
    spin_lock(&list_lock);
    ...
    remove_element
    spin_unlock(&list_lock);
    ...
    call_rcu(&el->head, el_free);
    ...
}
```

```
5.
void el_free(struct rcu_head *rhp)
{
    release_referenced();
}
```

The key point is that the initial reference added by add() is not removed until after a grace period has elapsed following removal. This means that search_and_reference() cannot find this element, which means that the value of el->rc cannot increase. Thus, once it reaches zero, there are no readers that can or ever will be able to reference the element. The element can therefore safely be freed. This in turn guarantees that if any reader finds the element, that reader may safely acquire a reference without checking the value of the reference counter.

A clear advantage of the RCU-based pattern in listing C over the one in listing B is that any call to search_and_reference() that locates a given object will succeed in obtaining a reference to that object, even given a concurrent invocation of delete() for that same object. Similarly, a clear advantage of both listings B and C over listing A is that a call to delete() is not delayed even if there are an arbitrarily large number of calls to search_and_reference() searching for the same object that delete() was invoked on. Instead, all that is delayed is the eventual invocation of kfree(), which is usually not a problem on modern computer systems, even the small ones.

In cases where delete() can sleep, synchronize_rcu() can be called from delete(), so that el_free() can be subsumed into delete as follows:

```
4.
delete()
{
    spin_lock(&list_lock);
    ...
    remove_element
    spin_unlock(&list_lock);
    ...
    synchronize_rcu();
    if (atomic_dec_and_test(&el->rc))
        kfree(el);
    ...
}
```

As additional examples in the kernel, the pattern in listing C is used by reference counting of struct pid, while the pattern in listing B is used by struct posix_acl.