Using RCU hlist_nulls to protect list and objects

This section describes how to use hlist_nulls to protect read-mostly linked lists and objects using SLAB_TYPESAFE_BY_RCU allocations.

Please read the basics in Documentation/RCU/listRCU.rst

Using 'nulls'

Using special makers (called 'nulls') is a convenient way to solve following problem :

A typical RCU linked list managing objects which are allocated with SLAB_TYPESAFE_BY_RCU kmem_cache can use following algos :

1) Lookup algo

```
rcu_read_lock()
begin:
obj = lockless_lookup(key);
if (obj) {
  if (!try_get_ref(obj)) // might fail for free objects
    goto begin;
  /*
  * Because a writer could delete object, and a writer could
  * reuse these object before the RCU grace period, we
  * must check key after getting the reference on object
  */
  if (obj->key != key) { // not the object we expected
    put_ref(obj);
    goto begin;
  }
}
rcu_read_unlock();
```

Beware that lockless_lookup(key) cannot use traditional hlist_for_each_entry_rcu() but a version with an additional memory barrier (smp_rmb())

```
lockless_lookup(key)
{
  struct hlist_node *node, *next;
  for (pos = rcu_dereference((head)->first);
      pos && ({ next = pos->next; smp_rmb(); prefetch(next); 1; }) &&
      ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; });
      pos = rcu_dereference(next))
    if (obj->key == key)
      return obj;
  return NULL;
}
```

And note the traditional hlist_for_each_entry_rcu() misses this smp_rmb():

```
struct hlist_node *node;
for (pos = rcu_dereference((head)->first);
      pos && ({ prefetch(pos->next); 1; }) &&
      ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1; });
      pos = rcu_dereference(pos->next))
 if (obj->key == key)
   return obj;
return NULL;
```

Quoting Corey Minyard:

```
"If the object is moved from one list to another list in-between the
time the hash is calculated and the next field is accessed, and the
object has moved to the end of a new list, the traversal will not
complete properly on the list it should have, since the object will
be on the end of the new list and there's not a way to tell it's on a
new list and restart the list traversal. I think that this can be
solved by pre-fetching the "next" field (with proper barriers) before
checking the key."
```

2) Insert algo

We need to make sure a reader cannot read the new 'obj->obj_next' value and previous value of 'obj->key'. Or else, an item could be deleted from a chain, and inserted into another chain. If new chain was empty before the move, 'next' pointer is NULL, and lockless reader can not detect it missed following items in original chain.

```
/*
* Please note that new inserts are done at the head of list,
* not in the middle or end.
*/
obj = kmem_cache_alloc(...);
lock_chain(); // typically a spin_lock()
obj->key = key;
/*
* we need to make sure obj->key is updated before obj->next
* or obj->refcnt
*/
smp_wmb();
atomic_set(&obj->refcnt, 1);
hlist_add_head_rcu(&obj->obj_node, list);
unlock_chain(); // typically a spin_unlock()
```

3) Remove algo

Nothing special here, we can use a standard RCU hlist deletion. But thanks to SLAB_TYPESAFE_BY_RCU, beware a deleted object can be reused very very fast (before the end of RCU grace period)

```
if (put_last_reference_on(obj) {
  lock_chain(); // typically a spin_lock()
  hlist_del_init_rcu(&obj->obj_node);
  unlock_chain(); // typically a spin_unlock()
  kmem_cache_free(cachep, obj);
}
```

Avoiding extra smp_rmb()

With hlist_nulls we can avoid extra smp_rmb() in lockless_lookup() and extra smp_wmb() in insert function.

For example, if we choose to store the slot number as the 'nulls' end-of-list marker for each slot of the hash table, we can detect a race (some writer did a delete and/or a move of an object to another chain) checking the final 'nulls' value if the lookup met the end of chain. If final 'nulls' value is not the slot number, then we must restart the lookup at the beginning. If the object was moved to the same chain, then the reader doesn't care : It might eventually scan the list again without harm.

1) lookup algo

```
head = &table[slot];
rcu_read_lock();
begin:
hlist_nulls_for_each_entry_rcu(obj, node, head, member) {
  if (obj->key == key) {
    if (!try_get_ref(obj)) // might fail for free objects
      goto begin;
    if (obj->key != key) { // not the object we expected
      put_ref(obj);
      goto begin;
    }
  goto out;
}
/*
* if the nulls value we got at the end of this lookup is
* not the expected one, we must restart lookup.
* We probably met an item that was moved to another chain.
*/
if (get_nulls_value(node) != slot)
goto begin;
obj = NULL;

out:
rcu_read_unlock();
```

2) Insert function

```
/*
* Please note that new inserts are done at the head of list,
* not in the middle or end.
*/
obj = kmem_cache_alloc(cachep);
lock_chain(); // typically a spin_lock()
obj->key = key;
/*
* changes to obj->key must be visible before refcnt one
*/
smp_wmb();
```

```
atomic_set(&obj->refcnt, 1);
/*
* insert obj in RCU way (readers might be traversing chain)
*/
hlist_nulls_add_head_rcu(&obj->obj_node, list);
unlock_chain(); // typically a spin_unlock()
```