Waiting for `kfree_rcu()`

Paul E. McKenney
Uladzislau Rezki
Boqun Feng
Neeraj Upadhyay
Jason A. Donenfeld
Jakub Kicinski
Vlastimil Babka
June 12, 2024
June 13, 2024
July 3, 2024
August 8, 2024

Back in the old days, `rcu_barrier()` would block until all pre-existing callbacks were invoked, which included waiting for all memory previously passed to `kfree_rcu()` to be freed. Perhaps as early as 2019, `rcu_barrier()` was no longer guaranteed to wait for the freeing of `kfree_rcu()` memory. This was not a problem because only memory allocated via `kmalloc()`, `vmalloc()`, and friends could be passed to `kfree_rcu()`. Other memory, in particular, that obtained from `kmem_cache_alloc()`, had to be freed via explicit RCU callbacks queued using `call_rcu()`.

This last restriction has recently been lifted because now `kfree()` can free memory returned from `kmem_cache_alloc()`.

However, if a module creates a kmem_cache containing RCU-protected objects that are freed using `kfree_rcu()`, that module has no way to wait for all memory to be freed before passing that `kmem_cache` to `kmem_cache_destroy()`.

This document looks at ways of handling this situation.

Approaches

The following sections cover possible approaches, listing advantages and disadvantages.

Status Quo

Document the current state, which is that a module must use `call_rcu()` rather than `kfree_rcu()` on memory obtained from `kmem_cache_alloc()` if that module calls `kmem_cache_destroy()` on that `kmem_cache` structure. This works, but adds extra code to this use case, and the penalty for incorrect use of `kfree_rcu()` is subtle memory-corruption bugs. It is only to be hoped that we can do better.

However, in the short term, this is the world we live in.

`rcu_barrier()` Waits for `kfree_rcu()`

Revert back to the pre-2019 semantics in which `rcu_barrier()` waits for the freeing of `kfree_rcu()` memory,

This adds complexity, overhead, and latency to `rcu_barrier()` that is unnecessary in most use cases. This approach is nevertheless worth looking into, and can be obtained by adding to `rcu_barrier()` an invocation of the `kfree_rcu_barrier()` function described in the next section. The big advantage of this approach is that it allows RCU/slab users to get their jobs done without dealing with yet more API members, which should earn it a high score on the Rusty Scale.

`kfree_rcu_barrier()` Waits for `kfree_rcu()`

Add a new `kfree_rcu_barrier()` function that waits for the freeing of all memory that has previously been passed to `kfree_rcu()`.

The simplest known way to implement this is to add an `rcu_head` structure to the `kvfree_rcu_bulk_data` structure. A call to `kfree_rcu_barrier()` would then traverse lists of in-flight `kvfree_rcu_bulk_data` structures, passing them to `call_rcu()` along with a callback function that would free them. This would be followed by a call to `rcu_barrier()` that would wait for all this memory to be freed.

~~Except that this approach fails to account for the low-memory behavior of `kfree_rcu_mightsleep()`, which involves a call to `synchronize_rcu()` and then `kfree()`. This case can be handled using a new `srcu_struct`, along with an `srcu_read_lock()` preceding the `synchronize_rcu()` and an `srcu_read_unlock()` following the `kfree()`. Given this infrastructure, could then invoke `synchronize_srcu()` to wait on all in-flight low-memory invocations of `kfree_rcu_mightsleep()`.~~

The reason that the above paragraph has been struck out is that `rcu_barrier()` only waits for `call_rcu()` invocations that have returns *before* that call to `rcu_barrier()`. We will also apply this rule to `kfree_rcu_mightsleep()`, which means that in the low-memory case, the memory will already be freed. There is therefore no need to explicitly wait for such calls to `kfree_rcu_mightsleep()` to free their memory.

As described, there would need to be mutual exclusion (presumably a mutex) between concurrent calls to `kfree_rcu_barrier()`. The same sort of batching optimizations used in `rcu_barrier()` might also be useful for `kfree_rcu_barrier()`.

This is likely to work reasonably well. However, one potential drawback is that this approach waits on all `kfree_rcu()` memory when it really only needs to wait for that `kfree_rcu()` memory associated with the `kmem_cache` structure that is to be passed to `kmem_cache_destroy()` prior to module unloading. (Of course, this might well turn out to be an advantage if there are modules creating and destroying large numbers of slabs of RCU-protected objects.)

The following sections look at some possible approaches that wait only on this one `kmem_cache` structure. In theory. In practices, these later approaches will likely have the slab allocator call something like a `kfree_rcu_barrier()` in order to do the needed waiting.

`kmem_cache_destroy()` Lingers for `kfree_rcu()`

"Just make the slab allocator handle it!" Here, `kmem_cache_destroy()` checks for memory not yet freed, and if there is any, arranges to defer the actual slab deallocation until all memory is freed.

This approach's downsides include losing valuable memory-leak debugging in the non-RCU case. Note that fully evaluating the advantages and disadvantages of this and the remaining approaches requires assistance from the slab maintainers. This document currently simply lists them.

One way of preserving this debugging information is to splat if all of the slab's memory has not been freed within a reasonable timeframe, perhaps the same 21 seconds that causes an RCU CPU stall warning (perhaps augmented by well-timed checks invoked from the `kfree_rcu()` workings). Note also that this lingering destruction might benefit non-RCU synchronization mechanisms, including reference counters and hazard pointers.

This approach appears to be the current direction.

kmem_cache_destroy() Lingers for kfree_rcu() and rcu_barrier()

This is the same as "`kmem_cache_destroy()` Lingers for `kfree_rcu()`" above, except that in the `SLAB_TYPESAFE_BY_RCU` case, `kmem_cache_destroy()` also lingers for `rcu_barrier()`. This lingering for `rcu_barrier()` is currently done in a batched fashion, courtesy of 657dc2f97220 ("slab: remove synchronous rcu_barrier() call in memcg cache release path").

Going back to per-kmem_cache_destroy() synchronous calls to rcu_barrier() would likely disappoint the cgroups use cases that motivated the change to use batching.

kmem_cache_destroy_rcu() Lingers for kfree_rcu()

"Just make the slab allocator supply another API to handle it!" Here, there is a new kmem_cache_destroy_rcu() that acts as described in the preceding section so that the original kmem_cache_destroy() function can retain its memory-leak debugging functionality.

kmem_cache_free_barrier() Waits for kfree_rcu()

Add a kmem_cache_free_barrier() that has roughly the same semantics as does kfree_rcu_barrier(), but is confined to the specified slab. Again, the original kmem_cache_destroy() function can retain its memory-leak debugging functionality.

kmem_cache_destroy_wait() Waits for kfree_rcu()

Add a kmem_cache_destroy_wait() that waits for all memory in the specified slab to be freed, then destroys that slab. Given a kmem_cache_free_barrier(), this could be implemented as follows:

```
kmem_cache_free_barrier(myslab);
kmem_cache_destroy(myslab);
```

kmem_cache_destroy_rcu/_barrier()

The idea here is to provide a asynchronous kmem_cache_destroy_rcu() as described above along with a kmem_cache_destroy_barrier() that waits for the destruction of all prior kmem_cache instances previously passed to kmem_cache_destroy_rcu(). Alternatively, could return a cookie that could be passed into a later call to kmem_cache_destroy_barrier(). This alternative has the advantage of isolating which kmem_cache instance is suffering the memory leak.

SLAB_DESTROY_ONCE_FULLY_FREED

Instead of adding a new kmem_cache_destroy_rcu() or kmem_cache_destroy_wait() API member, instead add a SLAB_DESTROY_ONCE_FULLY_FREED flag that can be passed to the existing kmem_cache_create() function. Use of this flag would suppress any warnings that would otherwise be issued by a call to kmem_cache_destroy() on the resulting kmem_cache if there was still slab memory yet to be freed, and it would also spawn workqueues (or timers or whatever) to do any needed cleanup work.

Your Ideas Here!!!