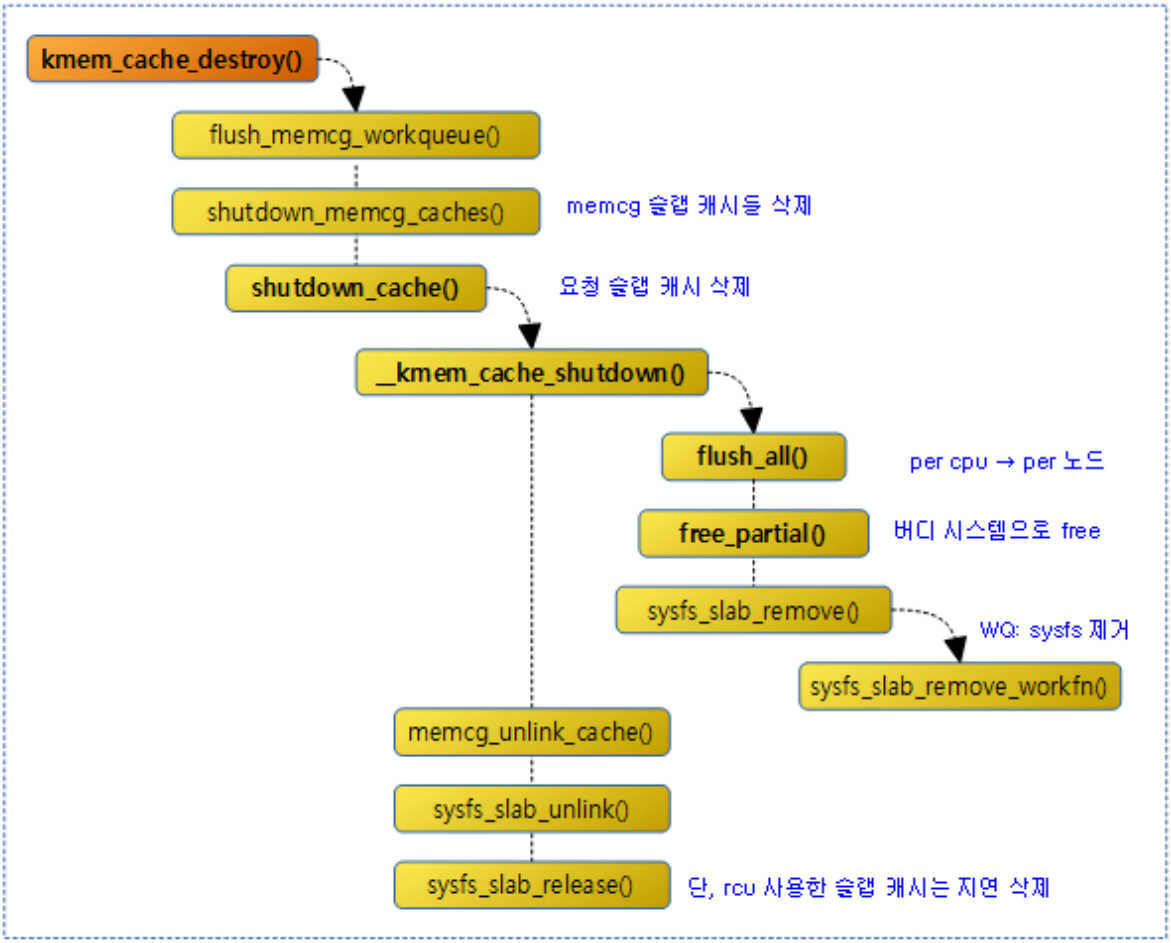
Slub Memory Allocator -11- (Clear Cache)

2016-06-07 (http://jake.dothome.co.kr/slub-cache-destroy/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Deleting the Slab Cache

The following illustration shows the process of clearing the slub cache.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/kmem_cache_destroy-1b.png)

kmem_cache_destroy()

mm/slab_common.c

```
01 void kmem_cache_destroy(struct kmem_cache *s)
02 {
03         int err;
04
05         if (unlikely(!s))
06                 return;
07
08         flush_memcg_workqueue(s);
```

```
09
10          get_online_cpus();
11          get_online_mems();
12
13          mutex_lock(&slab_mutex);
14
15          s->refcount--;
16          if (s->refcount)
17                  goto out_unlock;
18
19          err = shutdown_memcg_caches(s);
20          if (!err)
21                  err = shutdown_cache(s);
22
23          if (err) {
24                  pr_err("kmem_cache_destroy %s: Slab cache still has obje
    cts\n",
25                         s->name);
26                  dump_stack();
27          }
28  out_unlock:
29          mutex_unlock(&slab_mutex);
30
31          put_online_mems();
32          put_online_cpus();
33  }
34  EXPORT_SYMBOL(kmem_cache_destroy);
```

Delete the requested slab cache.

- In line 8 of the code, we flush the workqueue associated with the memcg slab cache.
- In line 15~17 of the code, decrement the reference counter of the slap cache and exit the function if it is not 0.
  - When creating and deleting the alias cache, only the reference counter should be decremented.
  - As soon as the reference counter reaches zero, the actual cache can be deleted.
- In line 19~21 of the code, shut down all the memcg caches in a loop. If it is processed without errors, all slab objects registered in the requested cache are freed, the per cpu and per node management structure is removed, and the cache with only the shell is added to the release list.

flush_memcg_workqueue()

mm/slab_common.c

```
01  static void flush_memcg_workqueue(struct kmem_cache *s)
02  {
03          mutex_lock(&slab_mutex);
04          s->memcg_params.dying = true;
05          mutex_unlock(&slab_mutex);
06
07          /*
08           * SLUB deactivates the kmem_caches through call_rcu. Make
09           * sure all registered rcu callbacks have been invoked.
10           */
11          if (IS_ENABLED(CONFIG_SLUB))
12                  rcu_barrier();
13
14          /*
15           * SLAB and SLUB create memcg kmem_caches through workqueue and
    SLUB
```

```
16          * deactivates the memcg kmem_caches through workqueue. Make sur
    e all
17          * previous workitems on workqueue are processed.
18          */
19         flush_workqueue(memcg_kmem_cache_wq);
20 }
```

memcg_kmem_cache_wq flush the workqueue for the memcg slab cache.

shutdown_memcg_caches()

mm/slab_common.c

```
01 static int shutdown_memcg_caches(struct kmem_cache *s)
02 {
03         struct memcg_cache_array *arr;
04         struct kmem_cache *c, *c2;
05         LIST_HEAD(busy);
06         int i;
07
08         BUG_ON(!is_root_cache(s));
09
10         /*
11          * First, shutdown active caches, i.e. caches that belong to onl
    ine
12          * memory cgroups.
13          */
14         arr = rcu_dereference_protected(s->memcg_params.memcg_caches,
15                                         lockdep_is_held(&slab_mutex));
16         for_each_memcg_cache_index(i) {
17                 c = arr->entries[i];
18                 if (!c)
19                         continue;
20                 if (shutdown_cache(c))
21                         /*
22                          * The cache still has objects. Move it to a tem
    porary
23                          * list so as not to try to destroy it for a sec
    ond
24                          * time while iterating over inactive caches bel
    ow.
25                          */
26                         list_move(&c->memcg_params.children_node, &bus
    y);
27                 else
28                         /*
29                          * The cache is empty and will be destroyed soo
    n. Clear
30                          * the pointer to it in the memcg_caches array s
    o that
31                          * it will never be accessed even if the root ca
    che
32                          * stays alive.
33                          */
34                         arr->entries[i] = NULL;
35         }
36
37         /*
38          * Second, shutdown all caches left from memory cgroups that are
    now
39          * offline.
40          */
41         list_for_each_entry_safe(c, c2, &s->memcg_params.children,
42                                  memcg_params.children_node)
43                 shutdown_cache(c);
44
```

```
45          list_splice(&busy, &s->memcg_params.children);
46
47          /*
48           * A cache being destroyed must be empty. In particular, this me
    ans
49           * that all per memcg caches attached to it must be empty too.
50           */
51          if (!list_empty(&s->memcg_params.children))
52                  return -EBUSY;
53          return 0;
54  }
```

Shutdown all memcg caches in the requested slab cache in a loop. Returns 0 if it succeeds, and -EBUSY if it fails.

shutdown_cache()

mm/slab_common.c

```
01  static int shutdown_cache(struct kmem_cache *s)
02  {
03          /* free asan quarantined objects */
04          kasan_cache_shutdown(s);
05
06          if (__kmem_cache_shutdown(s) != 0)
07                  return -EBUSY;
08
09          memcg_unlink_cache(s);
10          list_del(&s->list);
11
12          if (s->flags & SLAB_TYPESAFE_BY_RCU) {
13  #ifdef SLAB_SUPPORTS_SYSFS
14                  sysfs_slab_unlink(s);
15  #endif
16                  list_add_tail(&s->list, &slab_caches_to_rcu_destroy);
17                  schedule_work(&slab_caches_to_rcu_destroy_work);
18          } else {
19  #ifdef SLAB_SUPPORTS_SYSFS
20                  sysfs_slab_unlink(s);
21                  sysfs_slab_release(s);
22  #else
23                  slab_kmem_cache_release(s);
24  #endif
25          }
26
27          return 0;
28  }
```

Delete the requested slab cache and associated sysfs

- In line 4 of code, deallocate the data related to kasan.
- Delete the slab cache requested in line 6~7 of the code.
- Remove the link for memcg in the slab cache from line 9 of code.
- In line 10 of code, remove the slab cache from the global slab list.
- In lines 12~25 of the code, unlink and remove the sysfs-related directory of the slab cache. Slap caches using RCU are removed from the slab_caches_to_rcu_destroy_workfn() function via RCU after adding slab caches to the &slab_caches_to_rcu_destroy list, and removing those that are not immediately removed.
    - “/sys/kernel/slab/<slab 명>

__kmem_cache_shutdown()

mm/slub.c

```
1  /*
2   * Release all resources used by a slab cache.
3   */
```

```
01 int __kmem_cache_shutdown(struct kmem_cache *s)
02 {
03         int node;
04         struct kmem_cache_node *n;
05
06         flush_all(s);
07         /* Attempt to free all objects */
08         for_each_kmem_cache_node(s, node, n) {
09                 free_partial(s, n);
10                 if (n->nr_partial || slabs_node(s, node))
11                         return 1;
12         }
13         sysfs_slab_remove(s);
14         return 0;
15 }
```

Releases all slab pages in the requested slab cache and retrieves them to the buddy system.

- Move the per-CPU slab pages of the slab cache requested in line 6 to an n->partial list.
- Free the slap cache requested in code lines 8~12 and retrieve all of them to the buddy system. However, if a slab object exists in use, it returns 1 as a failure.
- Remove the sysfs-related objects from the cache requested in line 13 via the work queue. The schedule work function that is operated through the work queue is the sysfs_slab_remove_workfn() function.
- In line 14 of the code, if the slab cache eviction was successful, it returns 0.

Regarding sysfs for slab cache

Removing sysfs for slab cache with RCU

slab_caches_to_rcu_destroy_workfn()

mm/slab_common.c

```
01 static void slab_caches_to_rcu_destroy_workfn(struct work_struct *work)
02 {
03         LIST_HEAD(to_destroy);
04         struct kmem_cache *s, *s2;
05
06         /*
07          * On destruction, SLAB_TYPESAFE_BY_RCU kmem_caches are put on t
   he
08          * @slab_caches_to_rcu_destroy list.  The slab pages are freed
09          * through RCU and and the associated kmem_cache are dereference
   d
10          * while freeing the pages, so the kmem_caches should be freed o
   nly
```

```
         * after the pending RCU operations are finished.  As rcu_barrier()
         * is a pretty slow operation, we batch all pending destructions
         * asynchronously.
         */
        mutex_lock(&slab_mutex);
        list_splice_init(&slab_caches_to_rcu_destroy, &to_destroy);
        mutex_unlock(&slab_mutex);

        if (list_empty(&to_destroy))
                return;

        rcu_barrier();

        list_for_each_entry_safe(s, s2, &to_destroy, list) {
#ifdef SLAB_SUPPORTS_SYSFS
                sysfs_slab_release(s);
#else
                slab_kmem_cache_release(s);
#endif
        }
}
```

When a slab cache using the RCU is deleted, this function is called via the RCU to remove the kObject for sysfs from the slab cache.

sysfs_slab_release()

mm/slub.c

```
void sysfs_slab_release(struct kmem_cache *s)
{
        if (slab_state >= FULL)
                kobject_put(&s->kobj);
}
```

Removing sysfs for slab cache using workqueue

sysfs_slab_remove_workfn()

mm/slub.c

```
static void sysfs_slab_remove_workfn(struct work_struct *work)
{
        struct kmem_cache *s =
                container_of(work, struct kmem_cache, kobj_remove_work);

        if (!s->kobj.state_in_sysfs)
                /*
                 * For a memcg cache, this may be called during
                 * deactivation and again on shutdown.  Remove only once.
                 * A cache is never shut down before deactivation is
                 * complete, so no need to worry about synchronization.
                 */
                goto out;

#ifdef CONFIG_MEMCG
        kset_unregister(s->memcg_kset);
#endif
        kobject_uevent(&s->kobj, KOBJ_REMOVE);
out:
```

```
        kobject_put(&s->kobj);
}
```

A function that is called via workqueue removes the sysfs directory of the slab cache.

memcg_unlink_cache()

mm/slab_common.c

```
static void memcg_unlink_cache(struct kmem_cache *s)
{
        if (is_root_cache(s)) {
                list_del(&s->root_caches_node);
        } else {
                list_del(&s->memcg_params.children_node);
                list_del(&s->memcg_params.kmem_caches_node);
        }
}
```

Remove the link for memcg in the slab cache.

sysfs_slab_unlink()

mm/slub.c

```
void sysfs_slab_unlink(struct kmem_cache *s)
{
        if (slab_state >= FULL)
                kobject_del(&s->kobj);
}
```

Remove the sysfs directory of the slab cache.

Slab Cache Node Iterator

for_each_kmem_cache_node()

mm/slab.h

```
/*
 * Iterator over all nodes. The body will be executed for each node that has
 * a kmem_cache_node structure allocated (which is true for all online nodes)
 */
```

```
#define for_each_kmem_cache_node(__s, __node, __n) \
        for (__node = 0; __node < nr_node_ids; __node++) \
                 if ((__n = get_node(__s, __node)))

#endif
```

Traversal all nodes in the slab cache (@__s).

- Output factor __node = <nid starting from 0>
- Output Argument __n = kmem_cache_node Struct Pointer

get_node()

mm/slab.h

```
1 static inline struct kmem_cache_node *get_node(struct kmem_cache *s, int
  node)
2 {
3         return s->node[node];
4 }
```

Returns the kmem_cache_node information of the @node from the requested slab cache.

consultation

- Slab Memory Allocator -1- (Structure) (http://jake.dothome.co.kr/slub/) | Qc
- Slab Memory Allocator -2- (Initialize Cache) (http://jake.dothome.co.kr/kmem_cache_init) | Qc
- Slub Memory Allocator -3- (Create Cache) (http://jake.dothome.co.kr/slub-cache-create) | Qc
- Slub Memory Allocator -4- (Calculate Order) (http://jake.dothome.co.kr/slub-order) | Qc
- Slub Memory Allocator -5- | (http://jake.dothome.co.kr/slub-slub-alloc) Qc
- Slub Memory Allocator -6- (Assign Object) (http://jake.dothome.co.kr/slub-object-alloc) | Qc
- Slub Memory Allocator -7- (Object Unlocked) (http://jake.dothome.co.kr/slub-object-free) | Qc
- Slub Memory Allocator -8- (Drain/Flash Cache) (http://jake.dothome.co.kr/slub-drain-flush-cache) | Qc
- Slub Memory Allocator -9- (Cache Shrink) (http://jake.dothome.co.kr/slub-cache-shrink) | Qc
- Slub Memory Allocator -10- | (http://jake.dothome.co.kr/slub-slub-free) Qc
- Slub Memory Allocator -11- (Clear Cache (http://jake.dothome.co.kr/slub-cache-destroy)) | Sentence C – Current post
- Slub Memory Allocator -12- (Debugging Slub) (http://jake.dothome.co.kr/slub-debug) | Qc
- Slub Memory Allocator -13- (slabinfo) (http://jake.dothome.co.kr/slub-slabinfo) | 문c

---

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

‹ Slub Memory Allocator -5- (Slub allocation) (http://jake.dothome.co.kr/slub-slub-alloc/)

Slub Memory Allocator -9- (cache shrink) › (http://jake.dothome.co.kr/slub-cache-shrink/)

Munc Blog (2015 ~ 2024)