Slub Memory Allocator -7- (Object Unlocked)

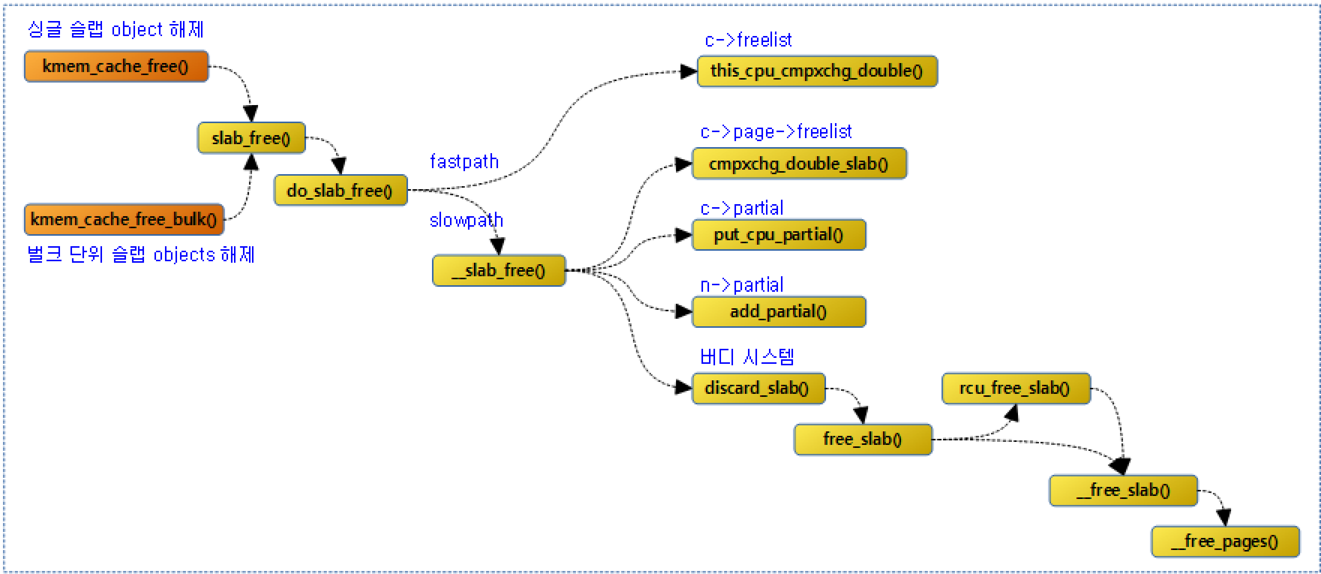
2016-06-07 (http://jake.dothome.co.kr/slub-object-free/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Deallocate a slap object

Release the slab object from the slub page of the specified kmem_cache.

The following illustration shows the flow of functions that are called when releasing a slab object.



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

Deallocate slab object

There are two types of APIs related to deallocating slab objects:

- Deallocate a single slap object
  - kmem_cache_free()
- Deallocate bulk unit slab object
  - kmem_cache_free_bulk()

kmem_cache_free()

mm/slub.c

```
void kmem_cache_free(struct kmem_cache *s, void *x)
{
```

```
3          s = cache_from_obj(s, x);
4          if (!s)
5                  return;
6          slab_free(s, virt_to_head_page(x), x, _RET_IP_);
7          trace_kmem_cache_free(_RET_IP_, x);
8  }
9  EXPORT_SYMBOL(kmem_cache_free);
```

deallocates a single-slab object.

- In line 3~5 of the code, we know the slab cache that the slab object points to. It is possible that the slab cache you know is the root cache. In other cases, the slab cache requested as an argument is still used.
- In line 6 of code, the slab object is free.

cache_from_obj()

mm/slab.h

```
01 static inline struct kmem_cache *cache_from_obj(struct kmem_cache *s, void *x)
02 {
03         struct kmem_cache *cachep;
04         struct page *page;
05 
06         /*
07          * When kmemcg is not being used, both assignments should return the
08          * same value. but we don't want to pay the assignment price in that
09          * case. If it is not compiled in, the compiler should be smart enough
10          * to not do even the assignment. In that case, slab_equal_or_root
11          * will also be a constant.
12          */
13         if (!memcg_kmem_enabled() &&
14             !unlikely(s->flags & SLAB_CONSISTENCY_CHECKS))
15                 return s;
16 
17         page = virt_to_head_page(x);
18         cachep = page->slab_cache;
19         if (slab_equal_or_root(cachep, s))
20                 return cachep;
21 
22         pr_err("%s: Wrong slab cache. %s but object is from %s\n",
23                __func__, cachep->name, s->name);
24         WARN_ON_ONCE(1);
25         return s;
26 }
```

Knows the slab cache that the slab object points to. It is possible that the slab cache you know is the root cache. Otherwise, it returns the requested slab cache as an argument.

- If you didn't enable memcg in lines 13~15 and didn't use the SLAB_DEBUG_FREE flag with a small probability, it will just return the given cache.
- In lines 17~20 of the code, if the slab cache that the slab page points to is the same as the slab cache requested as an argument, or if it is the root cache, it will return the slab cache that the slab page points to.

- If the cache specified in line 22~25 of the code is incorrect, it will output an error message and return the slab cache as it was requested as an argument.

deallocate slab object – Fastpath & Slowpath

slab_free()

mm/slub.c

```
static __always_inline void slab_free(struct kmem_cache *s, struct page *page,
                                      void *head, void *tail, int cnt,
                                      unsigned long addr)
{
        /*
         * With KASAN enabled slab_free_freelist_hook modifies the freelist
         * to remove objects, whose reuse must be delayed.
         */
        if (slab_free_freelist_hook(s, &head, &tail))
                do_slab_free(s, page, head, tail, cnt, addr);
}
```

Perform debug-related preprocessing routines before deallocating slab objects, and if there is no problem, deallocate slab objects from @head ~ @tail. If the @tail is null, only one @head is deallocated.

do_slab_free()

mm/slub.c

```
/*
 * Fastpath with forced inlining to produce a kfree and kmem_cache_free that
 * can perform fastpath freeing without additional function calls.
 *
 * The fastpath is only possible if we are freeing to the current cpu slab
 * of this processor. This typically the case if we have just allocated
 * the item before.
 *
 * If fastpath is not possible then fall back to __slab_free where we deal
 * with all sorts of special processing.
 *
 * Bulk free of a freelist with several objects (all pointing to the
 * same page) possible by specifying head and tail ptr, plus objects
 * count (cnt). Bulk free indicated by tail pointer being set.
 */
```

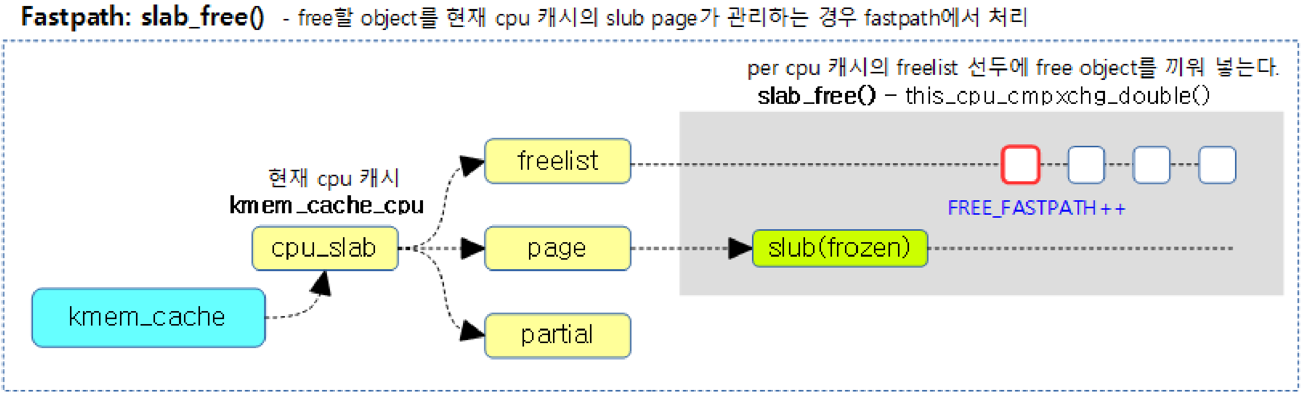
```
static __always_inline void do_slab_free(struct kmem_cache *s,
                                struct page *page, void *head, void *tail,
                                int cnt, unsigned long addr)
{
        void *tail_obj = tail ? : head;
        struct kmem_cache_cpu *c;
        unsigned long tid;
redo:
        /*
         * Determine the currently cpus per cpu slab.
```

```
11           * The cpu may change afterward. However that does not matter si
nce
12           * data is retrieved via this pointer. If we are on the same cpu
13           * during the cmpxchg then the free will succeed.
14           */
15          do {
16                  tid = this_cpu_read(s->cpu_slab->tid);
17                  c = raw_cpu_ptr(s->cpu_slab);
18          } while (IS_ENABLED(CONFIG_PREEMPT) &&
19                   unlikely(tid != READ_ONCE(c->tid)));
20
21          /* Same with comment on barrier() in slab_alloc_node() */
22          barrier();
23
24          if (likely(page == c->page)) {
25                  set_freepointer(s, tail_obj, c->freelist);
26
27                  if (unlikely(!this_cpu_cmpxchg_double(
28                                  s->cpu_slab->freelist, s->cpu_slab->tid,
29                                  c->freelist, tid,
30                                  head, next_tid(tid)))) {
31
32                          note_cmpxchg_failure("slab_free", s, tid);
33                          goto redo;
34                  }
35                  stat(s, FREE_FASTPATH);
36          } else
37                  __slab_free(s, page, head, tail_obj, cnt, addr);
38
39  }
```

Unassign slab objects from @head ~ @tail. If the @tail is null, only one @head is deallocated. (Fastpath, Slowpath)

- On line 8 of the code, the redo: label is. For fastpath purposes, this is where the transaction ID changes when attempting to deallocate an object from the per CPU cache, so it repeats again if it fails.
- In lines 15~19 of the code, the TID and Per-CPU caches are read atomically. At this point, preemption can be done at any time, so it is possible to switch tasks and come back during execution. Therefore, this routine performs a verification process behind the scenes to ensure that TID and cache are obtained from the same CPU.
- In line 22 of code, in order for slab's allocation/release algorithm to work without interrupt masks, it relies on the order in which cpu_slab data is read. In order to read tid before object and page, I used a compiler barrier to clearly distinguish the order of operations so that the compiler would not have to do optimization.
- In lines 24~35 of code, if the slappage is like c->page, add 1 or more objects to be inserted into the freelist lead of the per CPU cache, and increment the FREE_FASTPATH counter. If atomic substitution fails, go to the redo: label and try again. (Fastpath)
- In lines 36~37 of code, if the slab page is different from the page in the current per CPU cache, the object should be free in the slowpath way. (Slowpath call)

The following figure shows that the fastpath routine is activated when releasing an object, inserting only one free object at the beginning of the freelist of the current per CPU cache.



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

Deallocate slab object – Slowpath

__slab_free()

mm/slub.c -1/2-

```
/*
 * Slow path handling. This may still be called frequently since objects
 * have a longer lifetime than the cpu slabs in most processing loads.
 *
 * So we still attempt to reduce cache line usage. Just take the slab
 * lock and free the item. If there is no additional partial page
 * handling required then we can return immediately.
 */
```

```
static void __slab_free(struct kmem_cache *s, struct page *page,
                        void *head, void *tail, int cnt,
                        unsigned long addr)

{
        void *prior;
        int was_frozen;
        struct page new;
        unsigned long counters;
        struct kmem_cache_node *n = NULL;
        unsigned long uninitialized_var(flags);

        stat(s, FREE_SLOWPATH);

        if (kmem_cache_debug(s) &&
            !free_debug_processing(s, page, head, tail, cnt, addr))
                return;

        do {
                if (unlikely(n)) {
                        spin_unlock_irqrestore(&n->list_lock, flags);
                        n = NULL;
                }
                prior = page->freelist;
                counters = page->counters;
                set_freepointer(s, tail, prior);
                new.counters = counters;
                was_frozen = new.frozen;
                new.inuse -= cnt;
                if ((!new.inuse || !prior) && !was_frozen) {

                        if (kmem_cache_has_cpu_partial(s) && !prior) {

                                /*
```

```
35                                  * Slab was on no list before and will be
36                                  * partially empty
37                                  * We can defer the list move and instead
38                                  * freeze it.
39                                  */
40                                 new.frozen = 1;
41
42                         } else { /* Needs to be taken off a list */
43
44                                 n = get_node(s, page_to_nid(page));
45                                 /*
46                                  * Speculatively acquire the list_lock.
47                                  * If the cmpxchg does not succeed then we may
48                                  * drop the list_lock without any processing.
49                                  *
50                                  * Otherwise the list_lock will synchronize with
51                                  * other processors updating the list of slabs.
52                                  */
53                                 spin_lock_irqsave(&n->list_lock, flags);
54
55                         }
56                 }
57
58         } while (!cmpxchg_double_slab(s, page,
59                 prior, counters,
60                 head, new.counters,
61                 "__slab_free"));
```

Deallocate slab objects from @head ~ @tail (slowpath)

- Increment FREE_SLOWPATH counter at line 13 of code.
- If you used the SLAB_DEBUG_FLAGS flag in line 15~17 of the code, check it before releasing the object for slub object free debugging, and if there is a problem, it will print a warning message to let you know, stop processing, and leave the routine.
- This is where it will be repeated in line 19 if the atomic replacement fails.
- If a node is specified with a low probability in line 20~23 of the code, the node lock is released and the node designation is abandoned.
- On lines 24~29 of the code, prepare to insert the free object before page->freelist. The inuse counter decrements the number of objects to be free.
- In line 30~40 of code, if the slab page is not frozen (!was_frozen) and there are no free objects left in the c->freelist, prepare to change the slab page to a forzen state in order to be ready to add it to the c->partial list.
- In line 42~55 of the code, if the slab page is not frozen (!was_frozen) and all the objects on the slab page are in use, and the first free object occurs, it will know which node it belongs to and get a no-lock on it.
- In code lines 58~61, perform the atomic replacement as follows and repeat if it fails.
  - if page->freelist == prior && page->counters == counters
    - page->freelist = @head (the first of the objects to be terminated)
    - page->counters = new.counters

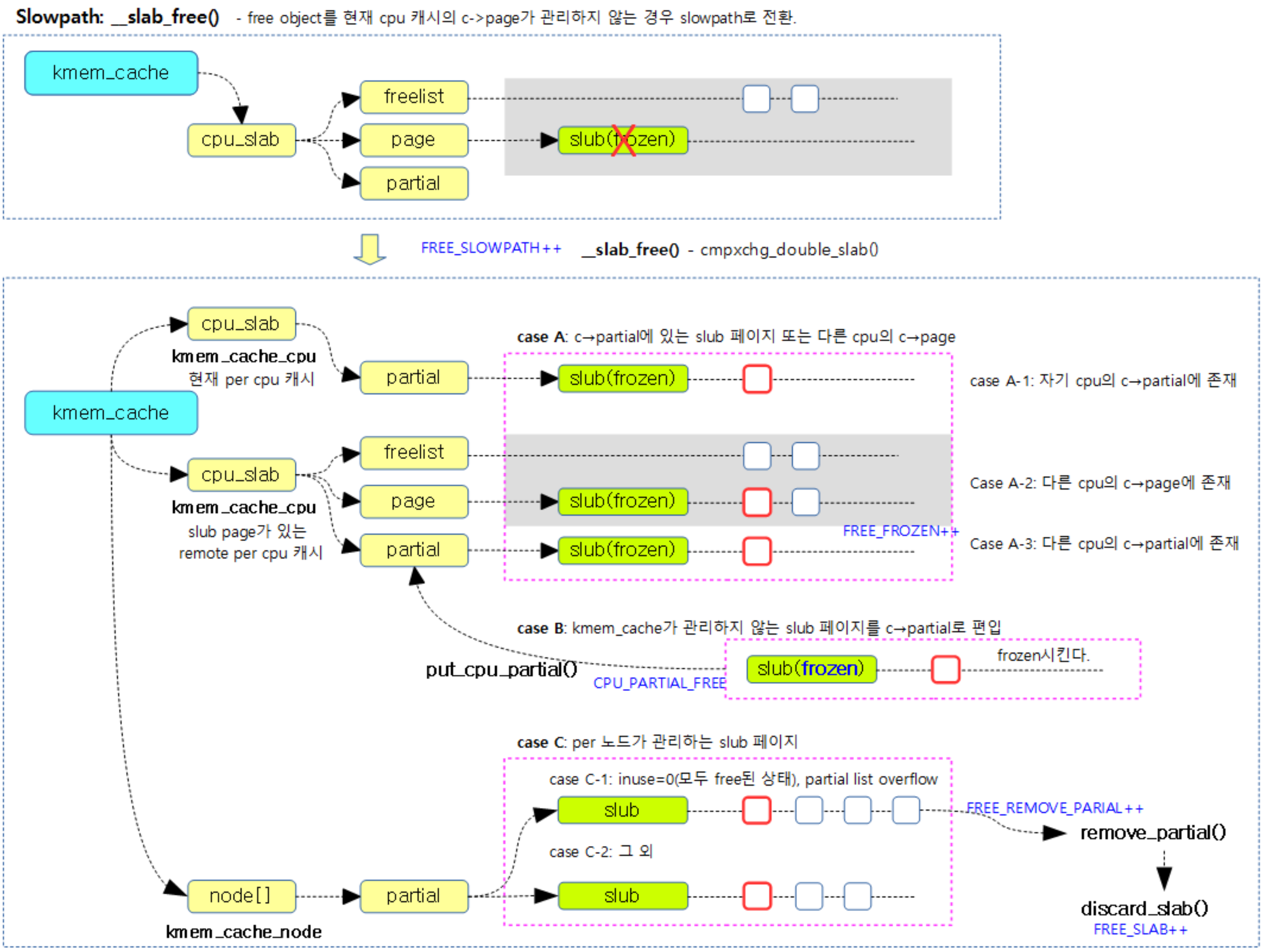
mm/slub.c -2/2-

```
01         if (likely(!n)) {
02 
03                 /*
04                  * If we just froze the page then put it onto the
05                  * per cpu partial list.
06                  */
07                 if (new.frozen && !was_frozen) {
08                         put_cpu_partial(s, page, 1);
09                         stat(s, CPU_PARTIAL_FREE);
10                 }
11                 /*
12                  * The list lock was not taken therefore no list
13                  * activity can be necessary.
14                  */
15                 if (was_frozen)
16                         stat(s, FREE_FROZEN);
17                 return;
18         }
19 
20         if (unlikely(!new.inuse && n->nr_partial >= s->min_partial))
21                 goto slab_empty;
22 
23         /*
24          * Objects left in the slab. If it was not on the partial list before
25          * then add it.
26          */
27         if (!kmem_cache_has_cpu_partial(s) && unlikely(!prior)) {
28                 if (kmem_cache_debug(s))
29                         remove_full(s, n, page);
30                 add_partial(n, page, DEACTIVATE_TO_TAIL);
31                 stat(s, FREE_ADD_PARTIAL);
32         }
33         spin_unlock_irqrestore(&n->list_lock, flags);
34         return;
35 
36 slab_empty:
37         if (prior) {
38                 /*
39                  * Slab on the partial list.
40                  */
41                 remove_partial(n, page);
42                 stat(s, FREE_REMOVE_PARTIAL);
43         } else {
44                 /* Slab must be on the full list */
45                 remove_full(s, n, page);
46         }
47 
48         spin_unlock_irqrestore(&n->list_lock, flags);
49         stat(s, FREE_SLAB);
50         discard_slab(s, page);
51 }
```

- This is the case when there is no need for access from code lines 1~18 to per node. After performing the following condition items, exit the function.
  - If it is a newly frozen slab page, add it to the c->partial list and increment the CPU_PARTIAL_FREE counter.
  - If it was previously frozen, increment the FREE_FROZEN counter.
- In line 20~21 of the code, if there is a low probability that all objects on the slab page are free objects, and the number of partial slabs in the node is overflowed, then move to the slab_empty: label to return the slab to the buddy system.

- In line 27~32 of the code, if the partial list is not supported in the per cpu cache and there is a low probability that there was not a single free object in the c->freelist, add it to the end of the n->partial list, and add it to the FREE_ADD_PARTIAL list.
- In code line 36, slab_empty: Labels. This is the label that will be moved to send the slab page consisting of only free objects to the buddy system.
- In line 37~42 of code, if there was an existing free object, remove it from the n->partial list and increment the FREE_REMOVE_PARTIAL counter.
- In line 43~46 of code, if there were no free objects in the past, remove them from the full list that was linked during SLUB debugging
- Unlock the node at lines 48~50 of code, increment the FREE_SLAB counter, and return the slab page to the buddy system.

The following illustration shows how to handle a slub object in the Slowpath step.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab_free-3a.png)

Add a specified slab page to c->partial

put_cpu_partial()

mm/slub.c

```
/*
 * Put a page that was just frozen (in __slab_free) into a partial page
 * slot if available.
 *
```

```
 * If we did not find a slot then simply move all the partials to the
 * per node partial list.
 */
```

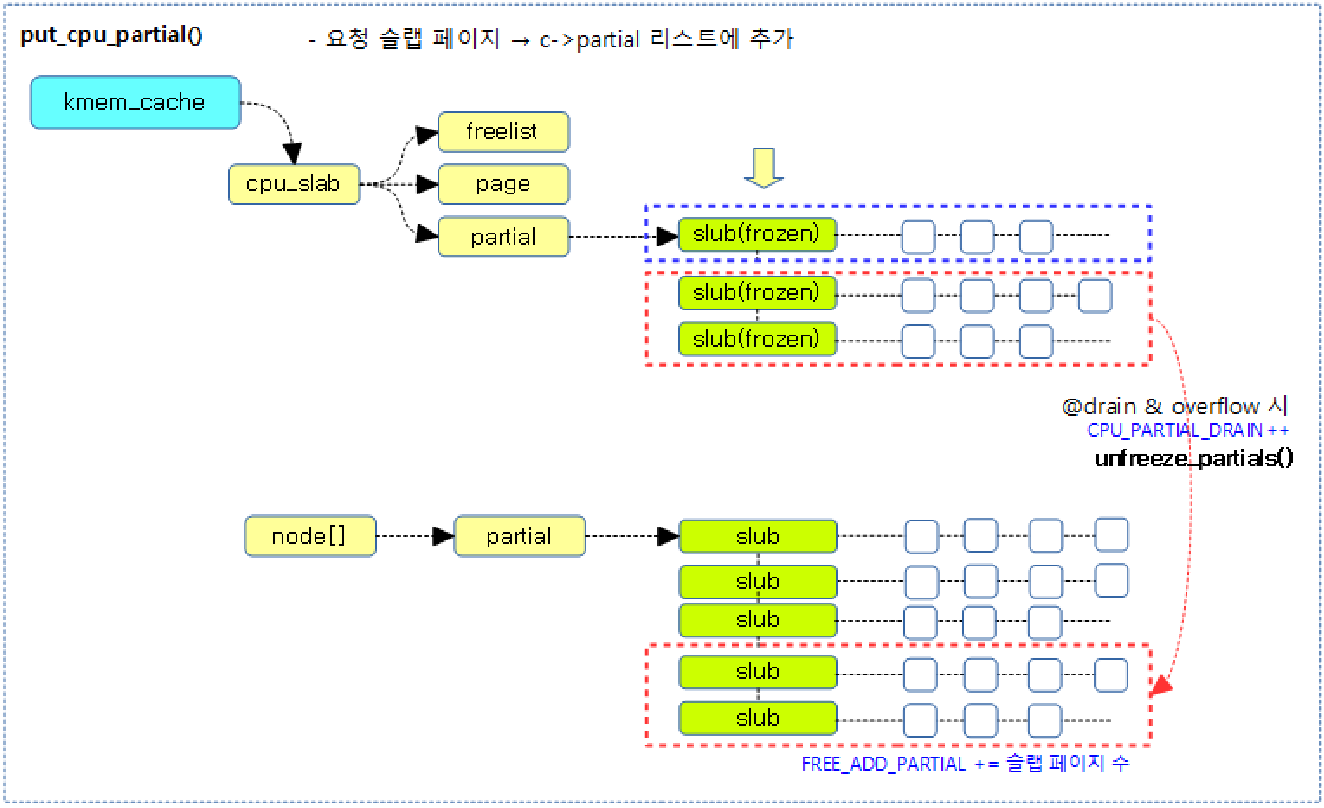
```
01 static void put_cpu_partial(struct kmem_cache *s, struct page *page, int drain)
02 {
03 #ifdef CONFIG_SLUB_CPU_PARTIAL
04         struct page *oldpage;
05         int pages;
06         int pobjects;
07 
08         preempt_disable();
09         do {
10                 pages = 0;
11                 pobjects = 0;
12                 oldpage = this_cpu_read(s->cpu_slab->partial);
13 
14                 if (oldpage) {
15                         pobjects = oldpage->pobjects;
16                         pages = oldpage->pages;
17                         if (drain && pobjects > s->cpu_partial) {
18                                 unsigned long flags;
19                                 /*
20                                  * partial array is full. Move the existing
21                                  * set to the per node partial list.
22                                  */
23                                 local_irq_save(flags);
24                                 unfreeze_partials(s, this_cpu_ptr(s->cpu_slab));
25                                 local_irq_restore(flags);
26                                 oldpage = NULL;
27                                 pobjects = 0;
28                                 pages = 0;
29                                 stat(s, CPU_PARTIAL_DRAIN);
30                         }
31                 }
32 
33                 pages++;
34                 pobjects += page->objects - page->inuse;
35 
36                 page->pages = pages;
37                 page->pobjects = pobjects;
38                 page->next = oldpage;
39 
40         } while (this_cpu_cmpxchg(s->cpu_slab->partial, oldpage, page)
41                                                                 != oldpage);
42         if (unlikely(!s->cpu_partial)) {
43                 unsigned long flags;
44 
45                 local_irq_save(flags);
46                 unfreeze_partials(s, this_cpu_ptr(s->cpu_slab));
47                 local_irq_restore(flags);
48         }
49         preempt_enable();
50 #endif
51 }
```

Add the specified srep page to the c->partial list. However, if the @drain is true, and the c->partial list is overflowed, all existing slab pages will be moved to the n->partial list.

- In line 8~41 of code, add a slap page to @c->partial with preemption disabled. page->pobjects is assigned by adding the pobject value of the existing slab page and the number of free objects on the current slab page. If @drain is true and the number of free objects in the slab cache is

overflowed, then if the c->partial list is overflowed, all the slab pages of the existing c->partial list will be moved to the n->partial list and the CPU_PARTIAL_DRAIN counter will be incremented.
- In lines 42~48 of code, if the number of restricted objects managed by the partial list in the per CPU cache is set to 0, all slab pages in the c->partial list are moved to the n->partial list

The following illustration shows adding a slab page to the top of a c->partial list.



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

Add a specified slab page to n->partial

add_partial()

mm/slub.c

```
static inline void add_partial(struct kmem_cache_node *n,
                                struct page *page, int tail)
{
        lockdep_assert_held(&n->list_lock);
        __add_partial(n, page, tail);
}
```

Adds the specified slab page to the specified position (lead or back) in the n->partial list.

__add_partial()

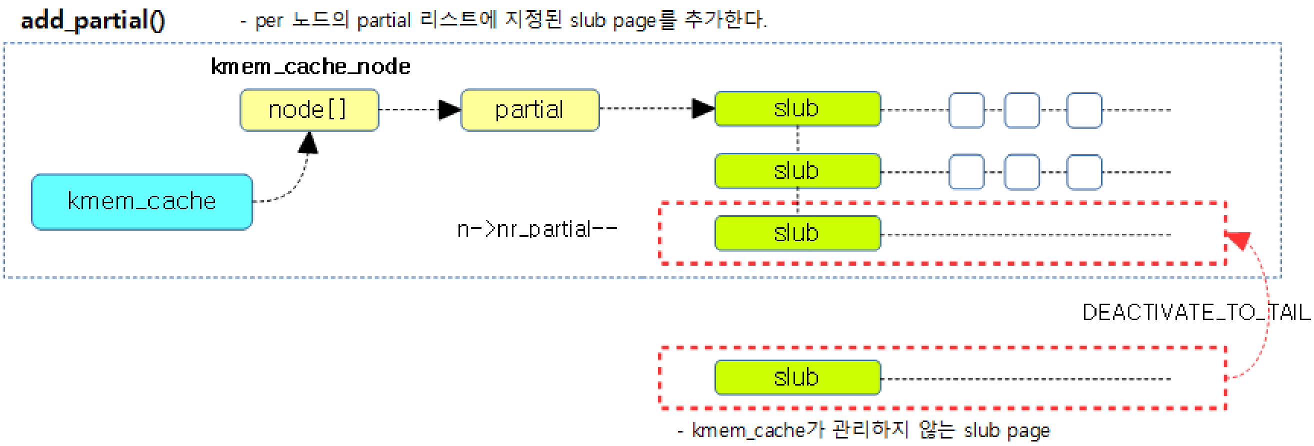
mm/slub.c

```
/*
 * Management of partially allocated slabs.
 */
```

```
static inline void
__add_partial(struct kmem_cache_node *n, struct page *page, int tail)
{
        n->nr_partial++;
        if (tail == DEACTIVATE_TO_TAIL)
                list_add_tail(&page->lru, &n->partial);
        else
                list_add(&page->lru, &n->partial);
}
```

Adds the specified slap page to the specified position (lead or back) in the n->partial list and increments the n->nr_partial.

The following image shows how a slub page is added to a specified position (at the front or back of the partial list) of the per node.



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

Remove slap pages specified by n->partial

remove_partial()

mm/slub.c

```
static inline void remove_partial(struct kmem_cache_node *n,
                                        struct page *page)
{
        lockdep_assert_held(&n->list_lock);
        __remove_partial(n, page);
}
```

Removes the specified slab page from the n->partial list.

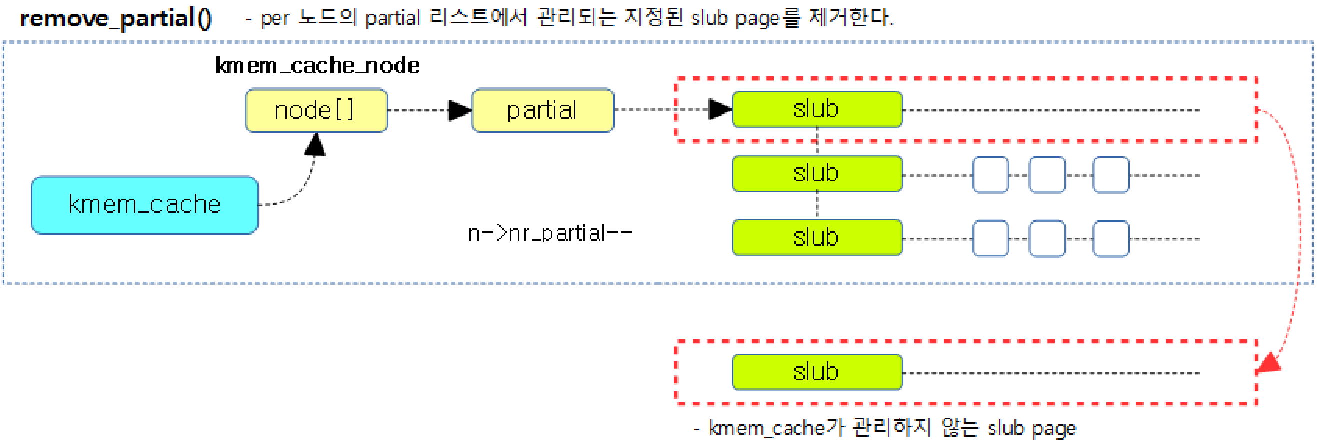
__remove_partial()

mm/slub.c

```
static inline void
__remove_partial(struct kmem_cache_node *n, struct page *page)
{
        list_del(&page->lru);
        n->nr_partial--;
}
```

Removes the specified slab page from the n->partial list, and decrements n->nr_partial.

The following image removes the per node from the partial list and replaces it with a state that is not managed by the kmem_cache.



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

All slab pages in c->freelist - > go to n->freelist

unfreeze_partials()

mm/slub.c

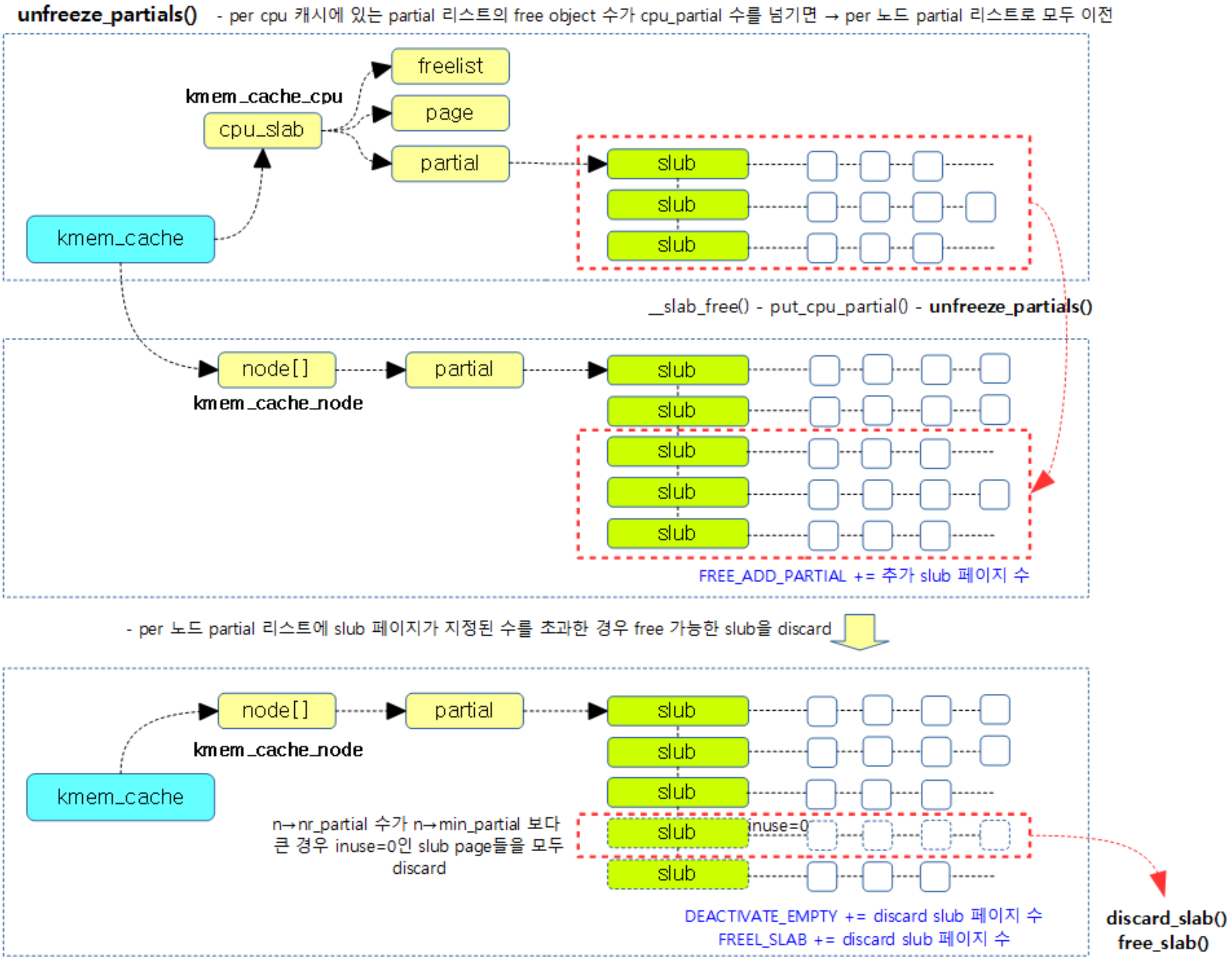
```
/*
 * Unfreeze all the cpu partial slabs.
 *
 * This function must be called with interrupts disabled
 * for the cpu using c (or some other guarantee must be there
 * to guarantee no concurrent accesses).
 */
```

```
static void unfreeze_partials(struct kmem_cache *s,
                struct kmem_cache_cpu *c)
{
#ifdef CONFIG_SLUB_CPU_PARTIAL
        struct kmem_cache_node *n = NULL, *n2 = NULL;
        struct page *page, *discard_page = NULL;

        while ((page = c->partial)) {
                struct page new;
                struct page old;

                c->partial = page->next;

                n2 = get_node(s, page_to_nid(page));
                if (n != n2) {
                        if (n)
                                spin_unlock(&n->list_lock);

                        n = n2;
                        spin_lock(&n->list_lock);
                }

                do {

                        old.freelist = page->freelist;
                        old.counters = page->counters;
```

```
				VM_BUG_ON(!old.frozen);

				new.counters = old.counters;
				new.freelist = old.freelist;

				new.frozen = 0;

			} while (!__cmpxchg_double_slab(s, page,
					old.freelist, old.counters,
					new.freelist, new.counters,
					"unfreezing slab"));

			if (unlikely(!new.inuse && n->nr_partial >= s->min_partial)) {
				page->next = discard_page;
				discard_page = page;
			} else {
				add_partial(n, page, DEACTIVATE_TO_TAIL);
				stat(s, FREE_ADD_PARTIAL);
			}
		}

		if (n)
			spin_unlock(&n->list_lock);

		while (discard_page) {
			page = discard_page;
			discard_page = discard_page->next;

			stat(s, DEACTIVATE_EMPTY);
			discard_slab(s, page);
			stat(s, FREE_SLAB);
		}
#endif
}
```

Add all slab pages managed by the c->partial list to the back of the n->partial list. If the n->partial list is overflowed, the slab pages that have been overflowed that do not have an object assigned to them will be returned to the buddy system.

The following figure shows the process of moving all the slub pages from the partial list of the per CPU cache to the partial list of the per node.



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

Slap Page - > Buddy System

discard_slab()

mm/slub.c

```
static void discard_slab(struct kmem_cache *s, struct page *page)
{
        dec_slabs_node(s, page_to_nid(page), page->objects);
        free_slab(s, page);
}
```

Dismiss the slap page and return it to the buddy system. It decrements n->nr_slabs (the number of slap pages) and decreases the n->total_objects value by the number of objects.

dec_slabs_node()

mm/slub.c

```
static inline void dec_slabs_node(struct kmem_cache *s, int node, int objects)
{
        struct kmem_cache_node *n = get_node(s, node);

        atomic_long_dec(&n->nr_slabs);
        atomic_long_sub(objects, &n->total_objects);
```

```
}
```

Decreases n->nr_slabs (number of slap pages) and decreases n->total_objects by the number of objects.

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) | Sentence C – Current post
- 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)) | Qc
- 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

---

4 thoughts to “Slub Memory Allocator -7- (Object 해제)”

IPARAN (HTTPS://WWW.BHRAL.COM/)
2021-12-03 13:07 (http://jake.dothome.co.kr/slub-object-free/#comment-306159)

Hello, Moon Young-il~ This is the 16th I'm Root Iparan~

Thank
you for leaving good materials and being able to have a good study!

Add
the slab page specified in the big picture slab_free-3a.png -> case B
c->partial in slab free slowpath ## put_cpu_partial()

** In the incoming part of the slub page that is not managed by the kmem_cache, the c->partial

Is the movement of a slub page that doesn't belong to the behavior of unfreeze_partials()?

I'm not sure where that part is, so can I ask~

RESPONSE (/SLUB-OBJECT-FREE/?REPLYTOCOM=306159#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-12-05 13:15 (http://jake.dothome.co.kr/slub-object-free/#comment-306167)

Hello?

On a system that uses a slub, all slab objects within a single slab page are allocated and fully in use, which is not managed by the slub system. Then, if any of those objects are free, you will have to manage them again in partial. By the way, there are also two partials, right? One on the CPU side and one on the Node side.

If all the slab objecs in the slab page are freed due to a slab object being freed, it will be moved to the partial list of nodes, but if it is not, it will be frozen(new.frozen) the page, and then use the cmpxchg_double_slab() function to free the slab object and then call the put_cpu_partial() function to add it to the partial list on that CPU.

And, as you asked, the further operation from the put_cpu_partial() function to the CPU partial list is performed by the this_cpu_cmpxchg() function.

The unfreeze_partials() function is called when it exceeds the maximum number managed by the CPU partial list.

I appreciate it.

RESPONSE (/SLUB-OBJECT-FREE/?REPLYTOCOM=306167#RESPOND)

IPARAN (HTTPS://WWW.BHRAL.COM/)
2021-12-05 21:24 (http://jake.dothome.co.kr/slub-object-free/#comment-306170)

Since all the objects are used, the pages that are not managed by the kmem_cahce will be added to the new CPU slub partial!
This way, you don't have to use spinlock + IRQ disable with list_lock members to add it to a node in partial!
Thanks for the details!

RESPONSE (/SLUB-OBJECT-FREE/?REPLYTOCOM=306170#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-12-06 09:46 (http://jake.dothome.co.kr/slub-object-free/#comment-306172)

Yes, and have a nice day!

RESPONSE (/SLUB-OBJECT-FREE/?REPLYTOCOM=306172#RESPOND)

LEAVE A COMMENT

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

댓글

이름 *

이메일 *

웹사이트

댓글 작성

Slub Memory Allocator -10- (Slub 해제) (http://jake.dothome.co.kr/slub-slub-free/)

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