Slub Memory Allocator -6- (Assign Object)

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

<kernel v5.0>

Slub Object Assignment (Fastpath, Slowpath)

Slub object assignments consist of four levels of slowpath assignments, starting with the fastest and the slowest at most.

Fastpath

Fastpath action atomically fetches the first object from the freelist of the per cpu slab cache. If there is no slab object to assign, move on to the Slowpath action.

- FastPath Steps:
  - Assign a free object on the fly from c->freelist
  - For reference, the abbreviation means:
    - s: kmem_cache struct pointing to the slab cache
    - C: S->cpu_slab of Per-CPU (kmem_cache_cpu structure)
    - n: s->node[] in per-node (kmem_cache_node struct)

Slowpath

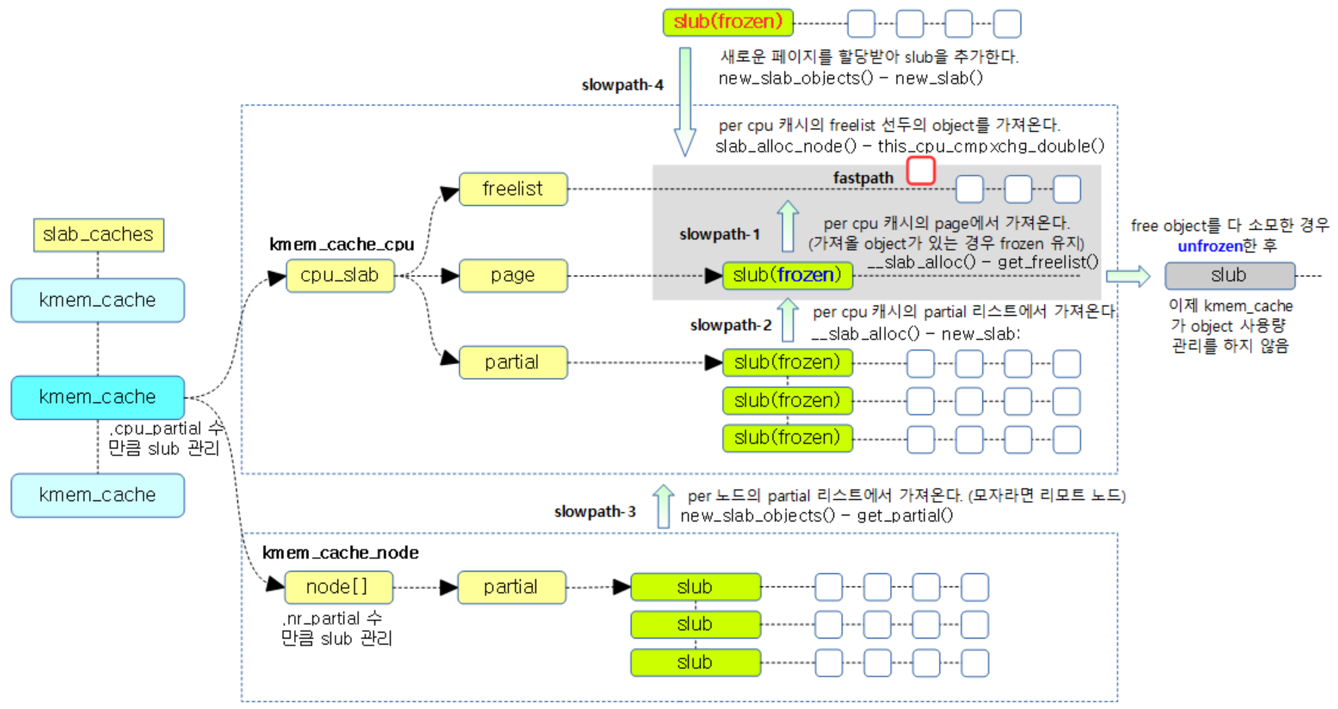
If c->freelist is empty and there is no object to retrieve, proceed with the slowpath operation as shown below, refill it, and then retry the fastpath operation.

- Slowpath Step 1:
  - c->page->freelist —(이동)—> c->freelist
  - Objects from the page->freelist of the per CPU cache are migrated to the freelist of the per CPU cache. If there is a free object, freeze the page. If there is no free object, proceed to step 2.
- Slowpath Step 2:
  - c->partial —(이동)—> c->page
  - Migrate the first slap page from the partial list of the per cpu cache to the page of the per cpu cache. If the partial list of per CPU cache is empty, proceed to step 3.
- Slowpath Step 3:
  - n->partial—(moves first slap page)—> c->page
  - n->partial—(moved a few places from the second slab page)—> c->partial
  - First, migrate the first slap page from the partial list of per nodes to a page in the per CPU cache.

- Second, starting with the next slap page in the partial list of per nodes, add it to the partial list of the per CPU cache.
  - However, if the number of free objects on the slab page exceeds half of the s->cpu_partial, only the slab page is migrated and the repeat is no more.
- If the partial list of per nodes is empty, it will retry from the partial list of other remote nodes. If even the other remote per nodes are unavailable (limiting the remote node's memory usage beyond a certain percentage), proceed to step 4.

- Slowpath Step 4:
  - Page allocator (buddy system)—(assign)—> c->page & c->freelist
  - After receiving the page assignment from the page allocator (buddy system) and initializing it as a slab page, it directly connects to c->page and connects the first free object to c->freelist.

The following illustration shows the priority of the slab object being assigned.


slub(frozen)
slowpath-4
새로운 페이지를 할당받아 slub을 추가한다.
new_slab_objects() – new_slab()
per cpu 캐시의 freelist 선두의 object를 가져온다.
slab_alloc_node() – this_cpu_cmpxchg_double()
fastpath
slab_caches
kmem_cache
kmem_cache_cpu
freelist
slowpath-1
per cpu 캐시의 page에서 가져온다.
(가져올 object가 있는 경우 frozen 유지)
__slab_alloc() – get_freelist()
free object를 다 소모한 경우
unfrozen한 후
slub
이제 kmem_cache
가 object 사용량
관리를 하지 않음
cpu_slab
page
slub(frozen)
slowpath-2
per cpu 캐시의 partial 리스트에서 가져온다.
__slab_alloc() – new_slab:
kmem_cache
partial
slub(frozen)
slub(frozen)
slub(frozen)
.cpu_partial 수
만큼 slub 관리
kmem_cache
slowpath-3
per 노드의 partial 리스트에서 가져온다. (모자라면 리모트 노드)
new_slab_objects() – get_partial()
kmem_cache_node
node[]
partial
slub
slub
slub
.nr_partial 수
만큼 slub 관리

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

Frozened Slub Page

The SLUB allocator manages the SLUB page from the per-cpu for fast allocation of SLUB objects without locks.

- cmpxchg double atomic operation to further delete the list.

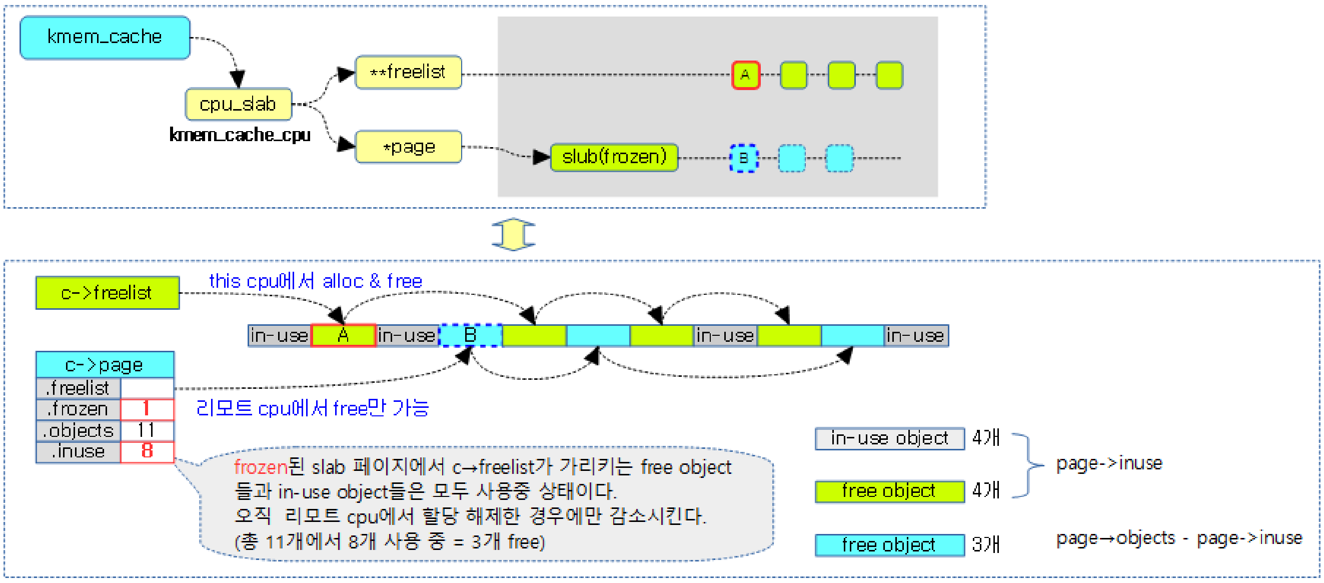
If page->frozen is set to 1, the freelist of the slub page is locked so that the current task manages it directly.

Manage two freelists

In fact, the management of the freelist is divided into two, and it works as follows.

- c->freelist
  - Currently, the CPU manages the freelist, so it is possible to allocate and deallocate slub objects (free).
- page->freelist
  - Other CPUs can only be freed here.

The following figure shows how the slub page is operated with 11 objects and fozen, and the freelist is managed in two.


kmem_cache
cpu_slab
kmem_cache_cpu
**freelist
*page
slub(frozen)
A
B
c->freelist
this cpu에서 alloc & free
in-use
A
B
c->page
.freelist
.frozen 1
.objects 11
.inuse 8
리모트 cpu에서 free만 가능
frozen된 slab 페이지에서 c->freelist가 가리키는 free object 들과 in-use object들은 모두 사용중 상태이다.
오직 리모트 cpu에서 할당 해제한 경우에만 감소시킨다.
(총 11개에서 8개 사용 중 = 3개 free)
in-use object 4개
free object 4개
page->inuse
free object 3개
page→objects - page->inuse

(http://jake.dothome.co.kr/wp-content/uploads/2016/06/frozen-slub-page-1c.png)

Inuse counter on the slub page

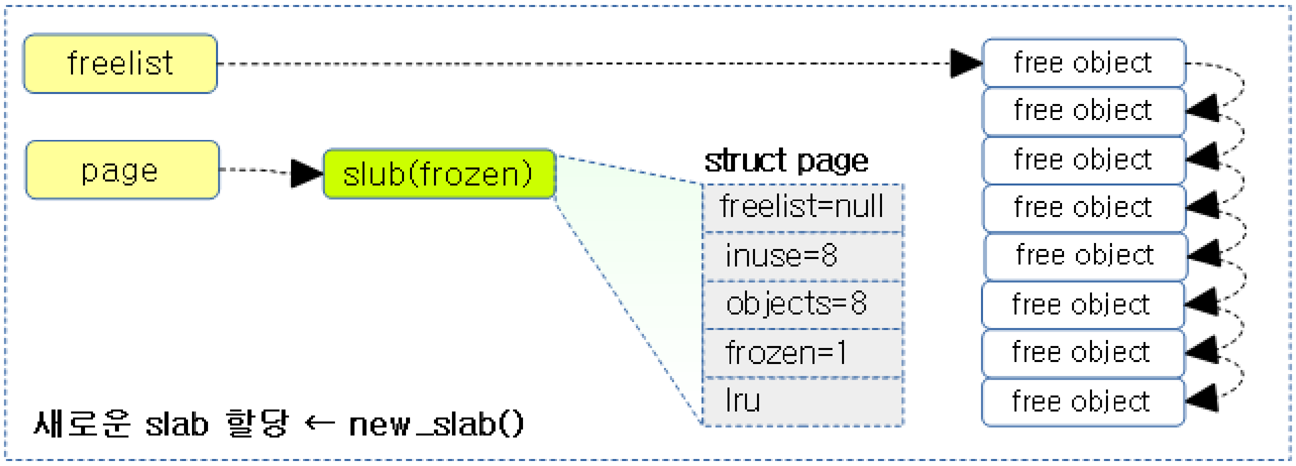
The inuse counter on the slub page indicates the number of objects in use, but also includes the free objects in the c->freelist as the number of objects in use. The changes to the inuse counter are as follows:

- Initial value
  - After the slab page is first assigned, all free objects are linked to C->freelist, so the number of page->objects is assigned to the page->inuse value to mark them as all in use.
- At the time of assignment
  - When you assign an object, there is no change to the page->inuse counter.
- On Release
  - When freed from this CPU, the page->inuse counter remains unchanged.
  - When disabling from the remote CPU, the page->inuse counter is decremented.
- On the Move
  - When moving from c->page->freelist to c->freelist, the page->inuse value is reflected as the page->objects value, and all objects are in use.

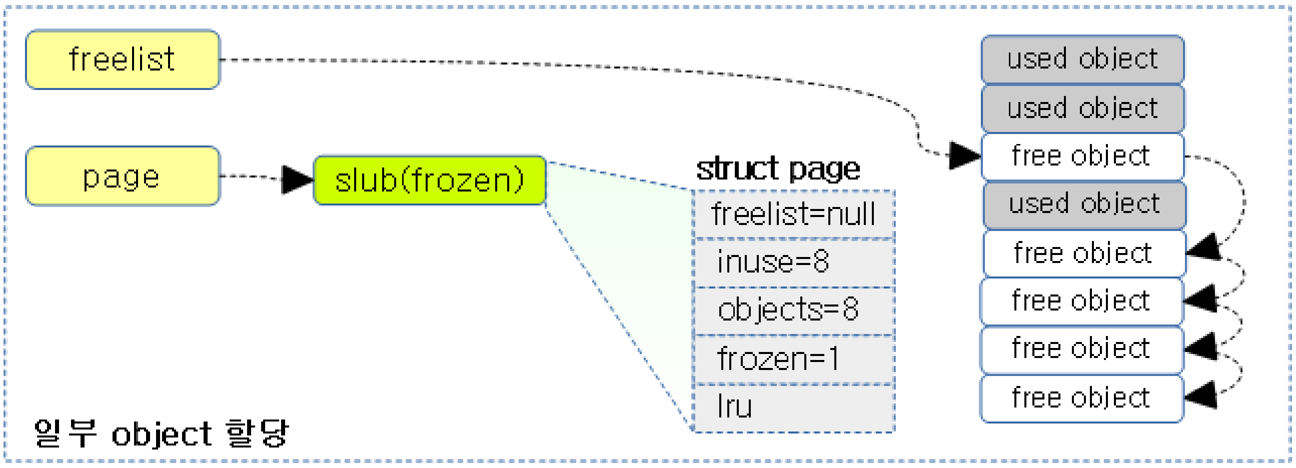
unfrozen state

Free objects in slab pages managed by n->partial are not included in the number of objects in use.

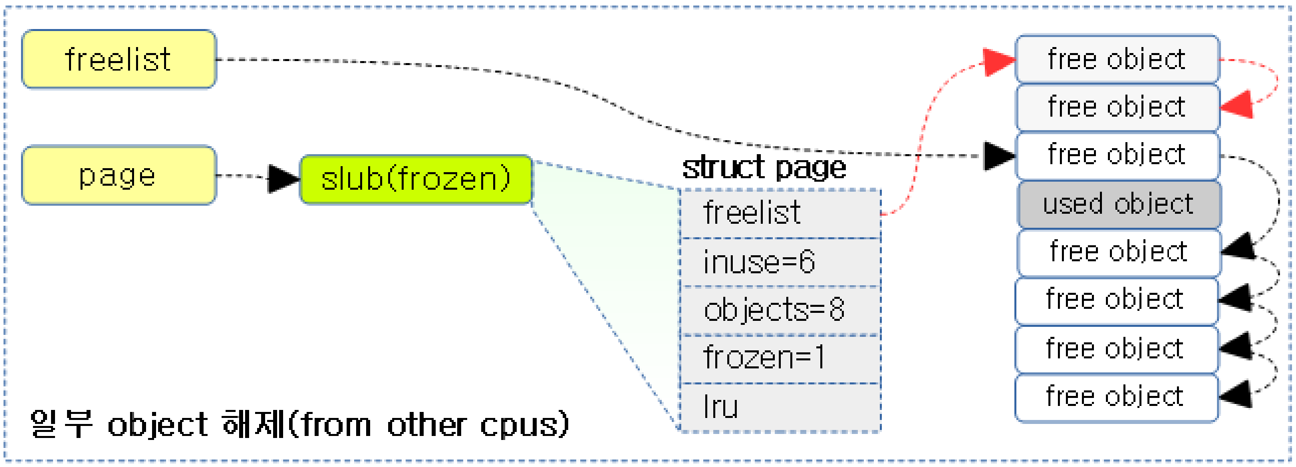
The following three figures show that the Page->inuse counter is decremented only when deallocating from the remote CPU.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/frozen-slub-page-3.png)



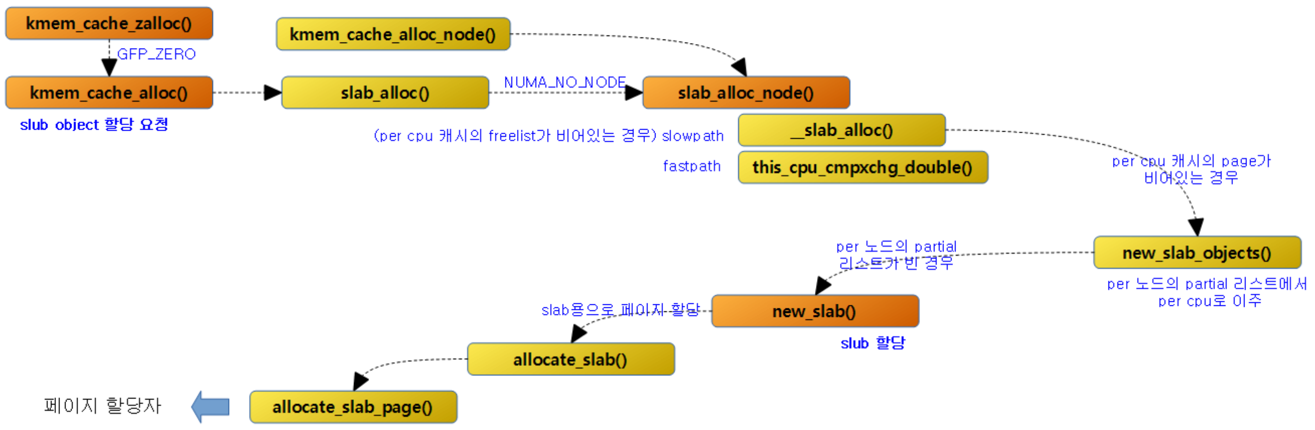
(http://jake.dothome.co.kr/wp-content/uploads/2016/06/frozen-slub-page-4.png)



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/frozen-slub-page-5a.png)

Assign a slab cache object

The following figure outlines the order in which the allocation request for a slub object is performed by function. If there are no objects to allocate by the last step, the Buddy system allocates a page to construct a slub and initializes the objects inside it.



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

kmem_cache_zalloc()

include/linux/slab.h

```
/*
 * Shortcuts
 */
```

```
static inline void *kmem_cache_zalloc(struct kmem_cache *k, gfp_t flags)
{
        return kmem_cache_alloc(k, flags | __GFP_ZERO);
}
```

It is allocated a slub object initialized to zero from the specified cache.

kmem_cache_alloc()

mm/slub.c

```
void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
{
        void *ret = slab_alloc(s, gfpflags, _RET_IP_);

        trace_kmem_cache_alloc(_RET_IP_, ret, s->object_size,
                                s->size, gfpflags);

        return ret;
}
EXPORT_SYMBOL(kmem_cache_alloc);
```

It is allocated a slub object from the specified cache.

- trace 문자열: “call_site=%lx ptr=%p bytes_req=%zu bytes_alloc=%zu gfp_flags=%s”

kmem_cache_alloc_node()

mm/slub.c

```
void *kmem_cache_alloc_node(struct kmem_cache *s, gfp_t gfpflags, int node)
{
        void *ret = slab_alloc_node(s, gfpflags, node, _RET_IP_);

        trace_kmem_cache_alloc_node(_RET_IP_, ret,
                                    s->object_size, s->size, gfpflags, node);

        return ret;
}
EXPORT_SYMBOL(kmem_cache_alloc_node);
```

GFP flag to allocate a slub object from the specified node.

slab_alloc()

mm/slub.c

```
static __always_inline void *slab_alloc(struct kmem_cache *s,
                gfp_t gfpflags, unsigned long addr)
{
        return slab_alloc_node(s, gfpflags, NUMA_NO_NODE, addr);
}
```

One slub object is assigned to the entire node from the specified cache.

Assign a slab object – Fastpath

slab_alloc_node() – slub

mm/slub.c

```
/*
 * Inlined fastpath so that allocation functions (kmalloc, kmem_cache_alloc)
 * have the fastpath folded into their functions. So no function call
 * overhead for requests that can be satisfied on the fastpath.
 *
 * The fastpath works by first checking if the lockless freelist can be used.
 * If not then __slab_alloc is called for slow processing.
 *
 * Otherwise we can simply pick the next object from the lockless free list.
 */
```

```
static __always_inline void *slab_alloc_node(struct kmem_cache *s,
                gfp_t gfpflags, int node, unsigned long addr)
{
        void **object;
        struct kmem_cache_cpu *c;
        struct page *page;
        unsigned long tid;

        s = slab_pre_alloc_hook(s, gfpflags);
        if (!s)
                return NULL;
```

```
redo:
        /*
         * Must read kmem_cache cpu data via this cpu ptr. Preemption is
         * enabled. We may switch back and forth between cpus while
         * reading from one cpu area. That does not matter as long
         * as we end up on the original cpu again when doing the cmpxchg.
         *
         * We should guarantee that tid and kmem_cache are retrieved on
         * the same cpu. It could be different if CONFIG_PREEMPT so we need
         * to check if it is matched or not.
         */
        do {
                tid = this_cpu_read(s->cpu_slab->tid);
                c = raw_cpu_ptr(s->cpu_slab);
        } while (IS_ENABLED(CONFIG_PREEMPT) &&
                 unlikely(tid != READ_ONCE(c->tid)));

        /*
         * Irqless object alloc/free algorithm used here depends on sequence
         * of fetching cpu_slab's data. tid should be fetched before anything
         * on c to guarantee that object and page associated with previous tid
         * won't be used with current tid. If we fetch tid first, object and
         * page could be one associated with next tid and our alloc/free
         * request will be failed. In this case, we will retry. So, no problem.
         */
        barrier();

        /*
         * The transaction ids are globally unique per cpu and per operation on
         * a per cpu queue. Thus they can be guarantee that the cmpxchg_double
         * occurs on the right processor and that there was no operation on the
         * linked list in between.
         */

        object = c->freelist;
        page = c->page;
        if (unlikely(!object || !node_match(page, node))) {
                object = __slab_alloc(s, gfpflags, node, addr, c);
                stat(s, ALLOC_SLOWPATH);
        } else {
                void *next_object = get_freepointer_safe(s, object);

                /*
                 * The cmpxchg will only match if there was no additional
                 * operation and if we are on the right processor.
                 *
                 * The cmpxchg does the following atomically (without lock
                 * semantics!)
                 * 1. Relocate first pointer to the current per cpu area.
                 * 2. Verify that tid and freelist have not been changed
                 * 3. If they were not changed replace tid and freelist
                 *
                 * Since this is without lock semantics the protection is only
65                  * against code executing on this cpu *not* from access by
66                  * other cpus.
67                  */
68                 if (unlikely(!this_cpu_cmpxchg_double(
69                                 s->cpu_slab->freelist, s->cpu_slab->tid,
70                                 object, tid,
71                                 next_object, next_tid(tid)))) {
72 
73                         note_cmpxchg_failure("slab_alloc", s, tid);
74                         goto redo;
75                 }
76                 prefetch_freepointer(s, next_object);
77                 stat(s, ALLOC_FASTPATH);
78         }
79 
80         if (unlikely(gfpflags & __GFP_ZERO) && object)
81                 memset(object, 0, s->object_size);
82 
83         slab_post_alloc_hook(s, gfpflags, 1, &object);
84 
85         return object;
86 }
```

It is allocated a slub object from the specified node in the specified cache.

- In line 9~11 of the code, we perform the pre-preparations for the slab object assignment, and if there is a problem, we give up the assignment. If the memcg cache is in effect, the memcg cache is obtained.
  - If there is a problem with the allocation of slab memory due to fault injection, the allocation is abandoned.
- On line 12 of the code, the redo: label is. For fastpath purposes, the transaction ID for that CPU has changed from the per CPU cache, and if it fails, it will repeat again.
- In lines 23~27 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, in this routine, the verification process is performed behind the scenes via atomic to ensure that TID and cache are obtained from the same CPU.
- In line 37 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 line 46~47 of the code, we get an object from the freelist of the per cpu cache, and we also get a page.
- In line 48~50 of the code, if there is a low probability that there is no slab object to allocate, or if the node of the slab page known from the current per CPU cache is different from the specified node, the object will be assigned via the slowpath operation. At this point, the ALLOC_SLOWPATH stat increases.
- If the object is successfully fetched from lines 51~52 of the code, the FP(Free Pointer) value, i.e., the address of the next object, is known.
- In lines 68~75 of the code, try to operate the fastpath operation of the slab object assignment. Atomically, the freelist points to the next slab object, and the tid is replaced with the next tid. If there is a small probability that it fails due to a change in the CPU or a change in the transaction ID, it will revert back to the redo label and retry until the slab object allocation succeeds.

  - If a task is preemption and resumed on a different node's CPU, the transaction ID may be changed by the CPU or by a different remote CPU. In this case, give up using the object obtained from the freelist and try again.
  - 참고: Per-cpu (atomic operations) (http://jake.dothome.co.kr/per-cpu-atomic) | 문c
- In line 76~77 of the code, the assignment of the slab object was successful with the fastpath operation. Prefetch the cacheline for the next free object, and increment the ALLOC_FASTPATH stat.
- If you use the GFP_ZERO flag with a small probability in line 80~81 of the code, clear the object to 0 by the size of the assigned object.
- In line 83 of code, the assigned completed state is notified to debug (kmemcheck, kmemleak, kasan) routines, etc., and to memcg.
- Returns the slab object assigned in line 85 of code.

Lock-less per-cpu list manipulation

In general, locks are used for list manipulation, but for fast list manipulation without using locks, it is often implemented with RCU or Per-CPU. Since the allocation and decommissioning of slab objec is frequently used in kernel code, fast performance is also required, so the method using per-cpu was chosen among the solutions that support lock-less implementation. If you implement using per-CPU, you typically dispose of interrupts in order to block preemption and interrupt intervention. However, the allocation and deallocation of slab objects is implemented with preemption and interrupts allowed for faster performance. Using this technique, it is necessary to ensure that there is no problem with list manipulation in order to handle the situation where the slab object is preempted during the allocation and deallocation process. Therefore, it is possible to safely handle list manipulation using atomic, which can be exchanged after comparison (cmpxchg), but if only the pointer of an object is compared, it may not be possible to handle it safely in the event of a special situation such as an ABA problem. This has led to the need for additional transaction IDs and features such as cpmxchg_double to compare and process two values at once, which are supported by the latest x86_64 and arm64-bit architectures.

this_cpu_cmpxchg()

- this_cpu_cmpxchg(v, old, new)
- Compare one v value with the old value and change it to a new value only if it is the same.

this_cpu_cmpxcgh_double()

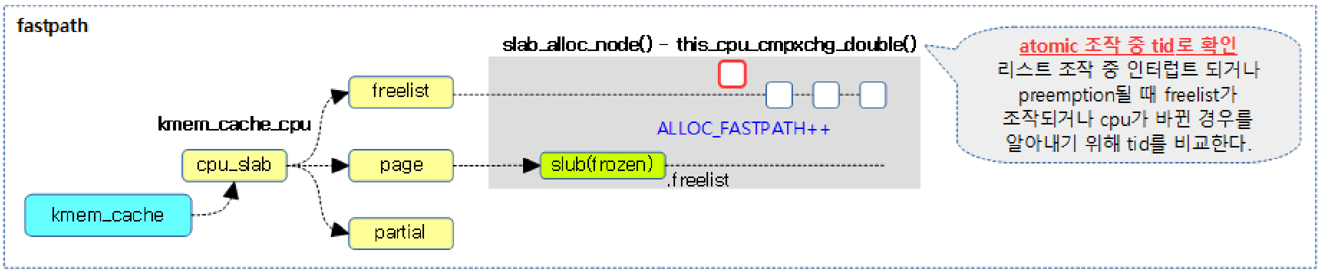
- this_cpu_cmpxcgh_double(v1, v2, old1, old2, new1, new2)
- Compare the two v1 and v2 values with the old1 and old2 values and change them to new1 and new2 values for the year, provided they are both the same.

Transaction ID

- Transaction IDs have different starting values on each CPU so that they don't overlap with each other, and use larger values to avoid intersecting incremental values.
- If you are using per-cpu with preemption enabled, it is safe to use the transaction id to check whether the cpu has changed, as the value of the currently accessed per-cpu may change due to changes in the cpu.

- If you try to change it with only the object address and not the transaction ID, you may have an ABA problem.
    - Consultation:
        - Multiprocessing and Multicore (2013) | Timothy Roscoe – Download PDF (https://www.systems.ethz.ch/sites/default/files/file/COURSES/2013_FALL_COURSES/2013_Fall_SPCA/lectures/19-Multiprocessing-1up.pdf)
        - Avoid the ABA problem when using lock-free algorithms (https://wiki.sei.cmu.edu/confluence/display/c/CON09-C.+Avoid+the+ABA+problem+when+using+lock-free+algorithms) | Carnegie Mellon University

The following figure shows how a slab object allocation request quickly allocates the first object from the freelist in the per CPU cache.


fastpath
slab_alloc_node() – this_cpu_cmpxchg_double()
freelist
kmem_cache_cpu
cpu_slab
page
partial
kmem_cache
ALLOC_FASTPATH++
slub(frozen)
.freelist
atomic 조작 중 tid로 확인
리스트 조작 중 인터럽트 되거나 preemption될 때 freelist가 조작되거나 cpu가 바뀐 경우를 알아내기 위해 tid를 비교한다.

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

slab_pre_alloc_hook()

mm/slub.c

```
01 static inline struct kmem_cache *slab_pre_alloc_hook(struct kmem_cache *s,
02                                                      gfp_t flags)
03 {
04         flags &= gfp_allowed_mask;
05 
06         fs_reclaim_acquire(flags);
07         fs_reclaim_release(flags);
08 
09         might_sleep_if(gfpflags_allow_blocking(flags));
10 
11         if (should_failslab(s, flags))
12                 return NULL;
13 
14         if (memcg_kmem_enabled() &&
15             ((flags & __GFP_ACCOUNT) || (s->flags & SLAB_ACCOUNT)))
16                 return memcg_kmem_get_cache(s);
17 
18         return s;
19 }
```

Perform the prerequisites for the slab object assignment, and if there is a problem, give up the assignment. It also returns the memcg cache if it is in effect.

- In line 4 of code, make sure that only the allowed GFP flags are used on early boot.
- If you're doing a direct-reclaim on line 9 of code, trigger the preemption point.

    - Sleep the current task and hand over the CPU to another task to handle requests with a higher priority than the current task.
- If you use the CONFIG_FAILSLAB kernel option in lines 11~12 of code, it will return null to give up the allocation if you are forced to commit a failure by the fault-injection function.
- In the case of a task with memcg applied in line 14~16 of code, find the per memcg cache and return it.

memcg_kmem_get_cache()

mm/memcontrol.c

```
/**
 * memcg_kmem_get_cache: select the correct per-memcg cache for allocation
 * @cachep: the original global kmem cache
 *
 * Return the kmem_cache we're supposed to use for a slab allocation.
 * We try to use the current memcg's version of the cache.
 *
 * If the cache does not exist yet, if we are the first user of it, we
 * create it asynchronously in a workqueue and let the current allocation
 * go through with the original cache.
 *
 * This function takes a reference to the cache it returns to assure it
 * won't get destroyed while we are working with it. Once the caller is
 * done with it, memcg_kmem_put_cache() must be called to release the
 * reference.
 */
```

```
struct kmem_cache *memcg_kmem_get_cache(struct kmem_cache *cachep)
{
        struct mem_cgroup *memcg;
        struct kmem_cache *memcg_cachep;
        int kmemcg_id;

        VM_BUG_ON(!is_root_cache(cachep));

        if (memcg_kmem_bypass())
                return cachep;

        memcg = get_mem_cgroup_from_current();
        kmemcg_id = READ_ONCE(memcg->kmemcg_id);
        if (kmemcg_id < 0)
                goto out;

        memcg_cachep = cache_from_memcg_idx(cachep, kmemcg_id);
        if (likely(memcg_cachep))
                return memcg_cachep;

        /*
         * If we are in a safe context (can wait, and not in interrupt
         * context), we could be be predictable and return right away.
         * This would guarantee that the allocation being performed
         * already belongs in the new cache.
         *
         * However, there are some clashes that can arrive from locking.
         * For instance, because we acquire the slab_mutex while doing
         * memcg_create_kmem_cache, this means no further allocation
         * could happen with the slab_mutex held. So it's better to
         * defer everything.
         */
        memcg_schedule_kmem_cache_create(memcg, cachep);
out:
        css_put(&memcg->css);
        return cachep;
}
```

If the current user task has memcg enabled, select the per memcg cache.

- If interrupted, current->mm is null, or if you are running as a kernel thread, do not change cachep.

slab_post_alloc_hook()

mm/slab.h

```
static inline void slab_post_alloc_hook(struct kmem_cache *s, gfp_t flag
s,
                                        size_t size, void **p)
{
        size_t i;

        flags &= gfp_allowed_mask;
        for (i = 0; i < size; i++) {
                p[i] = kasan_slab_alloc(s, p[i], flags);
                /* As p[i] might get tagged, call kmemleak hook after KA
SAN. */
                kmemleak_alloc_recursive(p[i], s->object_size, 1,
                                         s->flags, flags);
        }

        if (memcg_kmem_enabled())
                memcg_kmem_put_cache(s);
}
```

Perform a routine for post-performance in the slab object assignment. Perform debugging routines such as KASAN, kmemleak, etc.

node_match()

mm/slub.c

```
/*
 * Check if the objects in a per cpu structure fit numa
 * locality expectations.
 */
```

```
static inline int node_match(struct page *page, int node)
{
#ifdef CONFIG_NUMA
        if (!page || (node != NUMA_NO_NODE && page_to_nid(page) != nod
e))
                return 0;
#endif
        return 1;
}
```

In a UMA system, it always returns true, and in a NUMA system, it returns false if the node in page(slub) specified as an argument is different from the node specified as an argument.

- Using the preempt kernel of the NUMA system, the slub allocation routine should be preempted when attempting to allocate from the requested node, which will receive an object from the page (slub) of the current per cpu cache, and compare it to the node belonging to that page(slub) and return false in any other case so that it can be determined that the current cpu cache should not be used as is.

Free Pointer (FP)

prefetch_freepointer()

mm/slub.c

```
static void prefetch_freepointer(const struct kmem_cache *s, void *object)
{
        prefetch(object + s->offset);
}
```

Load the FP(Freelist Pointer) value of the requested slab object into the cache line in advance.

get_freepointer_safe()

mm/slub.c

```
static inline void *get_freepointer_safe(struct kmem_cache *s, void *object)
{
        unsigned long freepointer_addr;
        void *p;

        if (!debug_pagealloc_enabled())
                return get_freepointer(s, object);

        freepointer_addr = (unsigned long)object + s->offset;
        probe_kernel_read(&p, (void **)freepointer_addr, sizeof(p));
        return freelist_ptr(s, p, freepointer_addr);
}
```

It reads the FP(Freelist Pointer) value of the requested slab object and converts it to a pointer address.

get_freepointer()

mm/slub.c

```
static inline void *get_freepointer(struct kmem_cache *s, void *object)
{
        return freelist_dereference(s, object + s->offset);
}
```

Reads the FP (Freelist Pointer) value of the specified object and returns it to its address.

freelist_dereference()

mm/slub.c

```
/* Returns the freelist pointer recorded at location ptr_addr. */
static inline void *freelist_dereference(const struct kmem_cache *s,
                                        void *ptr_addr)
{
        return freelist_ptr(s, (void *)*(unsigned long *)(ptr_addr),
                            (unsigned long)ptr_addr);
}
```

Converts the @ptr_addr value, which is the FP (Freelist Pointer) value, into a pointer address and returns it.

freelist_ptr()

mm/slub.c

```
/*
 * Returns freelist pointer (ptr). With hardening, this is obfuscated
 * with an XOR of the address where the pointer is held and a per-cache
 * random number.
 */
```

```
static inline void *freelist_ptr(const struct kmem_cache *s, void *ptr,
                                 unsigned long ptr_addr)
{
#ifdef CONFIG_SLAB_FREELIST_HARDENED
        /*
         * When CONFIG_KASAN_SW_TAGS is enabled, ptr_addr might be tagged.
         * Normally, this doesn't cause any issues, as both set_freepointer()
         * and get_freepointer() are called with a pointer with the same tag.
         * However, there are some issues with CONFIG_SLUB_DEBUG code. For
         * example, when __free_slub() iterates over objects in a cache, it
         * passes untagged pointers to check_object(). check_object() in turns
         * calls get_freepointer() with an untagged pointer, which causes the
         * freepointer to be restored incorrectly.
         */
        return (void *)((unsigned long)ptr ^ s->random ^
                        (unsigned long)kasan_reset_tag((void *)ptr_addr));
#else
        return ptr;
#endif
}
```

Converts the @ptr_addr value, which is the FP (Freelist Pointer) value, into a pointer address and returns it.

- If you use the CONFIG_SLAB_FREELIST_HARDENED kernel options, the FP value is encoded along with a random value so that it is not recognizable.

Assign a Slap Object – Slowpath

__slab_alloc()

mm/slub.c

```
/*
 * Another one that disabled interrupt and compensates for possible
 * cpu changes by refetching the per cpu area pointer.
 */
```

```
static void *__slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int node,
                          unsigned long addr, struct kmem_cache_cpu *c)
{
        void *p;
        unsigned long flags;

        local_irq_save(flags);
#ifdef CONFIG_PREEMPT
        /*
         * We may have been preempted and rescheduled on a different
         * cpu before disabling interrupts. Need to reload cpu area
         * pointer.
         */
        c = this_cpu_ptr(s->cpu_slab);
#endif

        p = ___slab_alloc(s, gfpflags, node, addr, c);
        local_irq_restore(flags);
        return p;
}
```

With interrupts on the local CPU blocked, perform a slowpath routine for slab object assignment.

___slab_alloc()

mm/slub.c

```
/*
 * Slow path. The lockless freelist is empty or we need to perform
 * debugging duties.
 *
 * Processing is still very fast if new objects have been freed to the
 * regular freelist. In that case we simply take over the regular freelist
 * as the lockless freelist and zap the regular freelist.
 *
 * If that is not working then we fall back to the partial lists. We take the
 * first element of the freelist as the object to allocate now and move the
 * rest of the freelist to the lockless freelist.
 *
 * And if we were unable to get a new slab from the partial slab lists then
 * we need to allocate a new slab. This is the slowest path since it involves
 * a call to the page allocator and the setup of a new slab.
 *
 * Version of __slab_alloc to use when we know that interrupts are
 * already disabled (which is the case for bulk allocation).
 */
```

```
static void *___slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int node,
                          unsigned long addr, struct kmem_cache_cpu *c)
{
        void *freelist;
        struct page *page;

```

```c
        page = c->page;
        if (!page)
                goto new_slab;
redo:

        if (unlikely(!node_match(page, node))) {
                int searchnode = node;

                if (node != NUMA_NO_NODE && !node_present_pages(node))
                        searchnode = node_to_mem_node(node);

                if (unlikely(!node_match(page, searchnode))) {
                        stat(s, ALLOC_NODE_MISMATCH);
                        deactivate_slab(s, page, c->freelist, c);
                        goto new_slab;
                }
        }

        /*
         * By rights, we should be searching for a slab page that was
         * PFMEMALLOC but right now, we are losing the pfmemalloc
         * information when the page leaves the per-cpu allocator
         */
        if (unlikely(!pfmemalloc_match(page, gfpflags))) {
                deactivate_slab(s, page, c->freelist, c);
                goto new_slab;
        }

        /* must check again c->freelist in case of cpu migration or IRQ */
        freelist = c->freelist;
        if (freelist)
                goto load_freelist;

        freelist = get_freelist(s, page);

        if (!freelist) {
                c->page = NULL;
                stat(s, DEACTIVATE_BYPASS);
                goto new_slab;
        }

        stat(s, ALLOC_REFILL);

load_freelist:
        /*
         * freelist is pointing to the list of objects to be used.
         * page is pointing to the page from which the objects are obtained.
         * That page must be frozen for per cpu allocations to work.
         */
        VM_BUG_ON(!c->page->frozen);
        c->freelist = get_freepointer(s, freelist);
        c->tid = next_tid(c->tid);
        return freelist;

new_slab:

        if (slub_percpu_partial(c)) {
                page = c->page = slub_percpu_partial(c);
                slub_set_percpu_partial(c, page);
                stat(s, CPU_PARTIAL_ALLOC);
                goto redo;
        }

        freelist = new_slab_objects(s, gfpflags, node, &c);

        if (unlikely(!freelist)) {
```

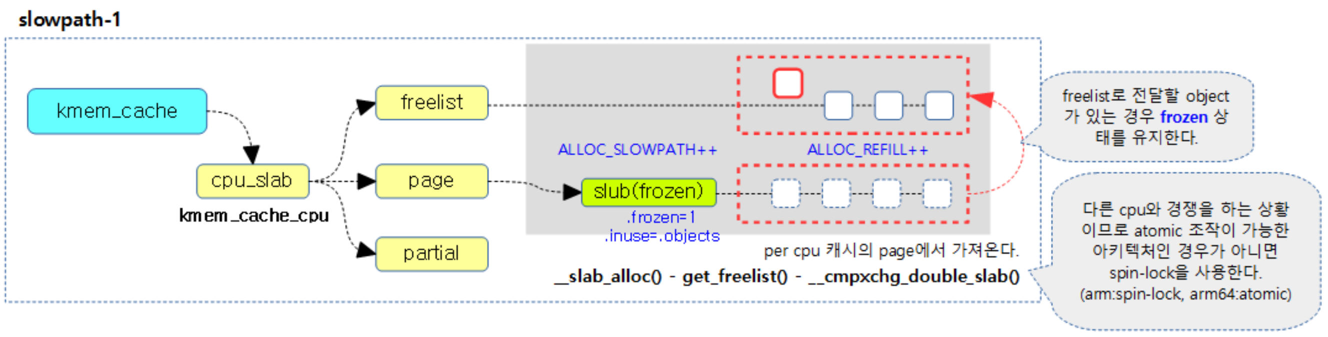
```
73                  slab_out_of_memory(s, gfpflags, node);
74                  return NULL;
75          }
76  
77          page = c->page;
78          if (likely(!kmem_cache_debug(s) && pfmemalloc_match(page, gfpflags)))
79                  goto load_freelist;
80  
81          /* Only entered in the debug case */
82          if (kmem_cache_debug(s) &&
83                          !alloc_debug_processing(s, page, freelist, addr))
84                  goto new_slab;  /* Slab failed checks. Next slab needed */
85  
86          deactivate_slab(s, page, get_freepointer(s, freelist), c);
87          return freelist;
88  }
```

Perform a step-by-step slowpath routine for slab object assignment. s->page->freelist –> s->partial –> n->partial steps and if that doesn't work, the last step is to allocate slab pages directly from the page allocator.

- If c->page is not specified in code lines 7~9, go to the new_slab to fetch a new slab page from the c->partial list.
- On line 10 of the code, the redo: label is. It is a label that is moved after charging a slab page from a c->partial or n->partial list to return to this label and retry the object assignment.
- In line 12~23 of the code, there is a low probability that the node in c->page and the node requested as an argument are different. It can also be a memoryless node configuration, so if the node in c->page is different from the adjacent node again, increment the ALLOC_NODE_MISMATCH counter, deactivate the slab page linked to s->page, assign null to page and freelist, and then go to the new_slab to get the new slab page from the c->partial list.
  - Normally, if a task is preemptioned and resched, and the task resumes from that location, it will be rescheduled to the original CPU or CPU belonging to that node, if possible, but if the CPU of that node is busy, it may be rescheduled to the CPU of another node. This is the case when the node on the current CPU may be different from the node that was originally requested.
- In line 30~33 of code, if the current request is not an emergency and an emergency slub page is selected, deactivate the slab page linked to s->page, assign null to page and freelist, and move to the new_slab to fetch a new slub page from the c->partial list.
  - If you are using the swap for NBD driver, use the ALLOC_NO_WATERMARKS flag when requesting a slab object assignment to allow TCP socket senders and receivers to take advantage of the emergency slub page as well.
- In lines 36~38 of the code, once again check that c->freelist is empty. If the object returned just before the interrupt is left in the freelist, it moves to the load_freelist label to receive the slab object from c->freelist.
- On line 40, move c->page->freelist to c->freelist. (SlowPath Step 1)
- In code lines 42~46, if there are no free slab objects in the c->freelist, the slab page is in use, so we assign null to c->page to quietly stop the slab page from being managed by the slab allocator, and increment the DEACTIVATE_BYPASS counter. Then go to the new_slab label to get a new slap page assigned. (This is the case when the first step of the slowpath failed.)

- In line 48 of the code, c->freelist is repopulated, so increment the ALLOC_REFILL counter. (This is the case if the Slowpath step 1 succeeds.)
- In code lines 50~59, load_freelist: Label. c->freelist points to the next free object, increments the transaction id, and returns the assigned slab object.
- In code line 61, new_slab: Label.
- In code lines 63~68, if a slab page exists in c->partial, only one slab page is moved to c->page. Then increment the CPU_PARTIAL_ALLOC counter and go to the redo: label to retry the allocation of the slab object. (SlowPath Step 2)
- On line 70 of the code, new_slab_objects you want to be allocated a new slab page from an n->partial list or buddy allocator, call the function to connect to c->page and c->freelist and get a new free object assigned. (Slowpath 3~4 steps)
- If a new slab object is not allocated in line 72~75, it will return null after OOM (Out Of Memory).
- In line 77~79 of the code, if all of the following situations are met with a high probability, go to the load_freelist label and retry the free object assignment.
  - When you don't need to debug Slub
  - The current request is not an emergency, but the emergency slub page is selected.
- If a debug request SLAB_CONSISTENCY_CHECKS in line 82~84 of code finds a problem in the consistency check, it moves to the new_slab label to be assigned a new slab page.
- In line 86 of code, deactivate the slab page linked to s->page and assign null to page and freelist.
- Returns the slab object assigned in line 87 of code.

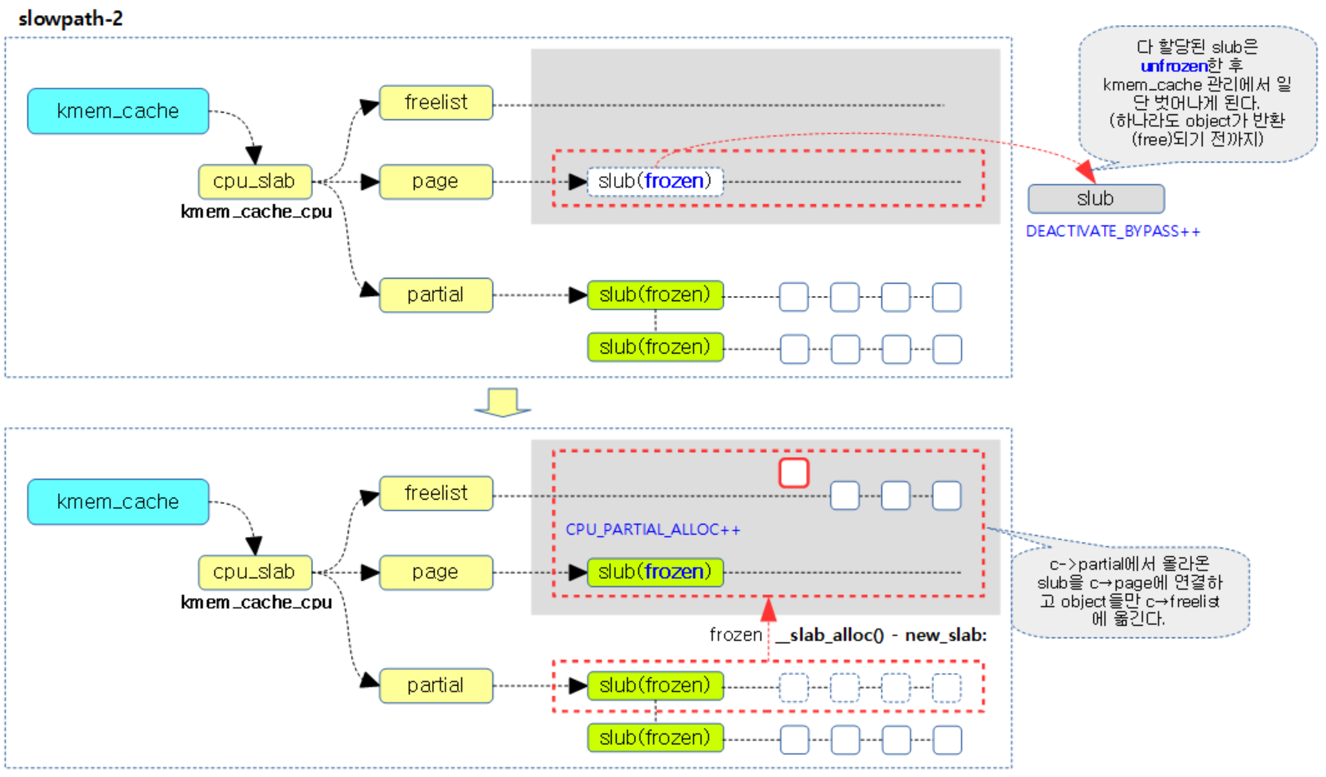
The following diagram shows the SlowPath step 1 process of moving from c->page->freelist to c->freelist>


slowpath-1
kmem_cache
cpu_slab
kmem_cache_cpu
freelist
page
partial
ALLOC_SLOWPATH++
ALLOC_REFILL++
slub(frozen)
.frozen=1
.inuse=.objects
per cpu 캐시의 page에서 가져온다.
__slab_alloc() - get_freelist() - __cmpxchg_double_slab()
freelist로 전달할 object가 있는 경우 frozen 상태를 유지한다.
다른 cpu와 경쟁을 하는 상황이므로 atomic 조작이 가능한 아키텍처인 경우가 아니면 spin-lock을 사용한다. (arm:spin-lock, arm64:atomic)

(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab_alloc_node-2c.png)

The following diagram shows the two-step Slowpath process of moving to c->partial – > c->page & c->freelist.

- Since all the objects on the slab page that were previously managed have been allocated, they are not directly managed by the slab cache allocator.



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

Slowpath Step 1 Processing

get_freelist()

mm/slub.c

```
/*
 * Check the page->freelist of a page and either transfer the freelist to the
 * per cpu freelist or deactivate the page.
 *
 * The page is still frozen if the return value is not NULL.
 *
 * If this function returns NULL then the page has been unfrozen.
 *
 * This function must be called with interrupt disabled.
 */
```

```
static inline void *get_freelist(struct kmem_cache *s, struct page *page)
{
        struct page new;
        unsigned long counters;
        void *freelist;

        do {
                freelist = page->freelist;
                counters = page->counters;

                new.counters = counters;
                VM_BUG_ON(!new.frozen);

                new.inuse = page->objects;
                new.frozen = freelist != NULL;

        } while (!__cmpxchg_double_slab(s, page,
                freelist, counters,
                NULL, new.counters,
                "get_freelist"));

        return freelist;
}
```

Returns the freelist of the page in the per CPU cache, and puts null into the freelist of the slub page, and changes it to All in use. (SlowPath Step 1)

- c->page->freelist – > When all free objects are moved to C->freelist, the value of S->Objects is assigned to the inuse containing the number of objects in use and changed to all of them in use.
- Assign null to page->freelist, and new.counters to page->counter, and repeat if that fails

---

Slowpath 3 & Step 4 Processing

new_slab_objects()

mm/slub.c

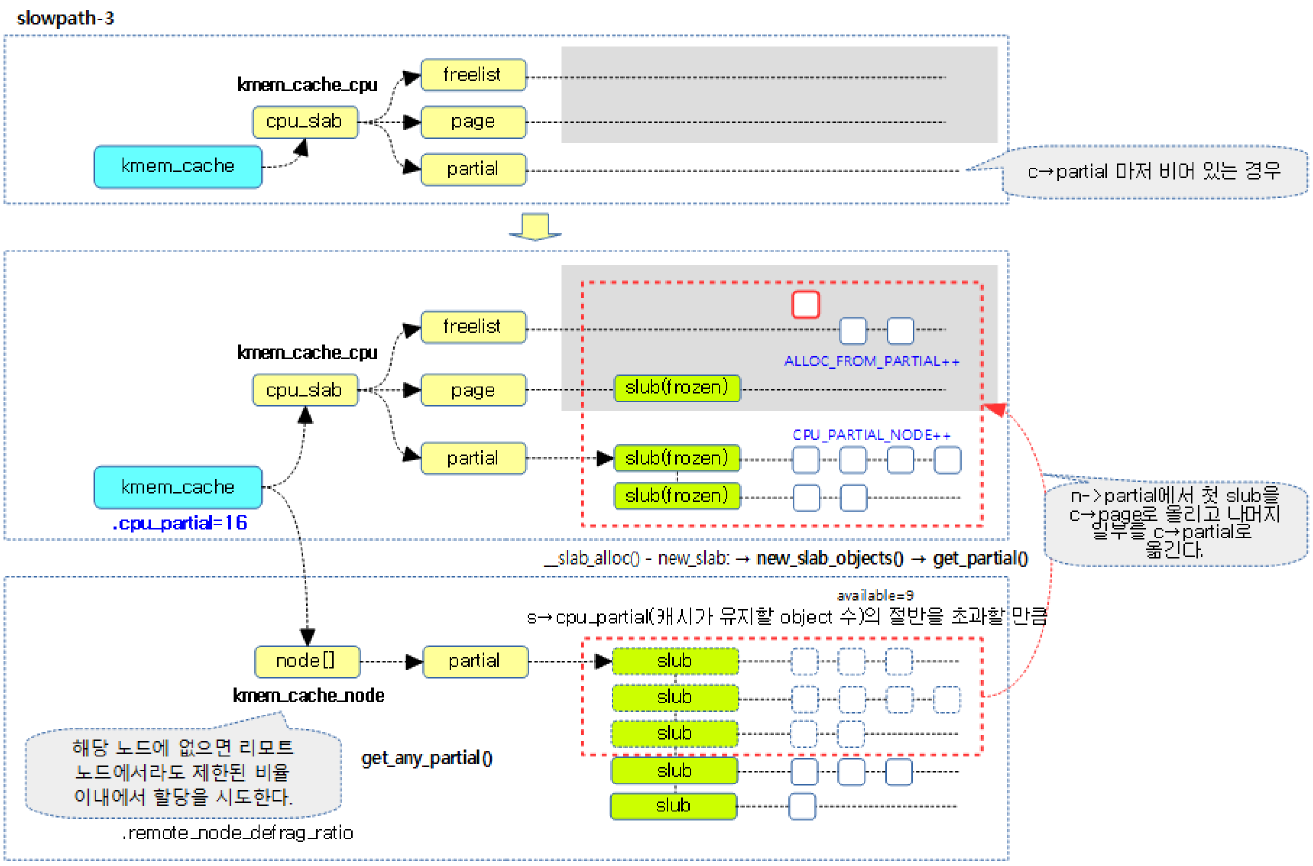
```
static inline void *new_slab_objects(struct kmem_cache *s, gfp_t flags,
                        int node, struct kmem_cache_cpu **pc)
{
        void *freelist;
        struct kmem_cache_cpu *c = *pc;
        struct page *page;

        WARN_ON_ONCE(s->ctor && (flags & __GFP_ZERO));
        freelist = get_partial(s, flags, node, c);

        if (freelist)
                return freelist;

        page = new_slab(s, flags, node);
        if (page) {
                c = raw_cpu_ptr(s->cpu_slab);
                if (c->page)
                        flush_slab(s, c);

                /*
                 * No other reference to the page yet so we can
                 * muck around with it freely without cmpxchg
                 */
                freelist = page->freelist;
                page->freelist = NULL;

                stat(s, ALLOC_SLAB);
                c->page = page;
                *pc = c;
        } else
                freelist = NULL;

        return freelist;
}
```

Move a few slab pages from the n->partial list to the c->freelist, c->page, and c->partial lists. If there is no slab page to move, a new slab page is assigned by the buddy system and assigned to c->freelist and c->page. If even page allocation fails, it returns null. (Slowpath 3~4 steps)

- In line 10~13 of the code, move a few slab pages from the partial list of local and remote nodes to the c->freelist, c->page, and c->partial lists, and return if there is a slab object to be assigned normally. (Slowpath Step 3)
- In line 15~30 of the code, a new page is assigned from the buddy system. It then connects to c->page, inserts nulls into c->page->freelist, and increments the ALLOC_SLAB counter. Since the CPU can be changed, the per-CPU cache is substituted in the output argument @pc. It then returns a freelist. (Slowpath Step 4)
- In line 31~32 of the code, the buddy system failed to allocate due to insufficient memory. NULL.

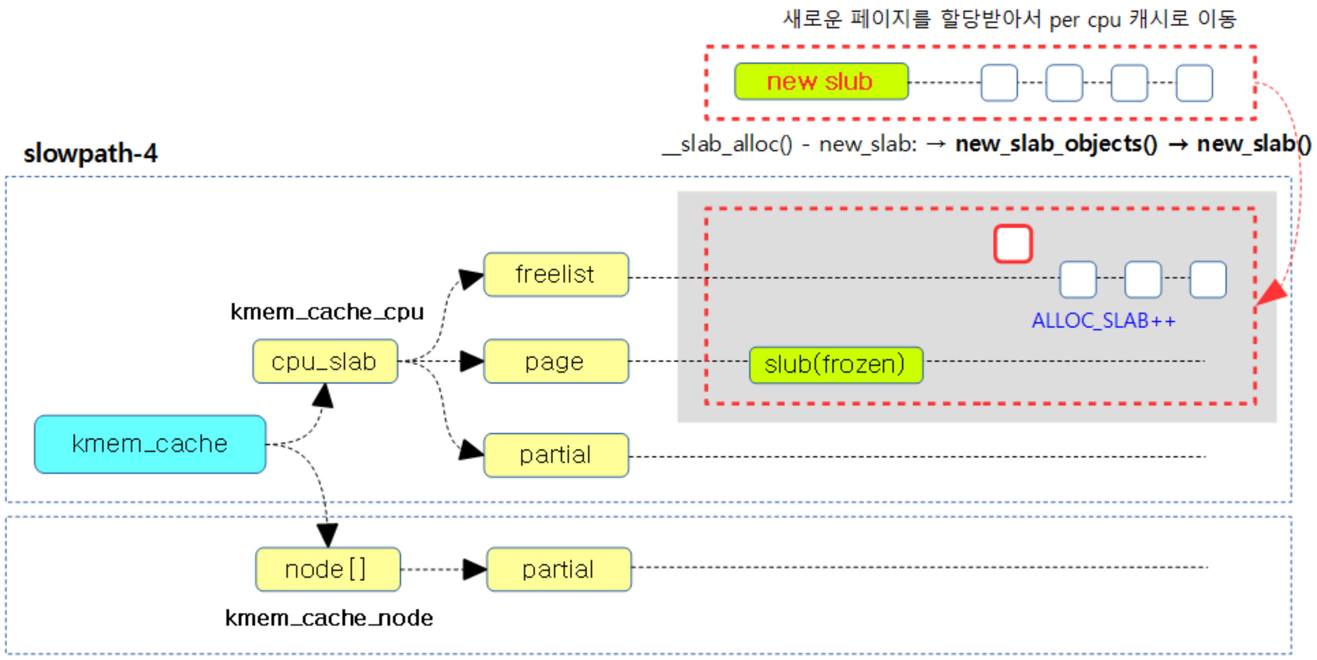
The following diagram shows the three-step slowpath process of moving to n->partial – > c->page & c->freelist & c->partial.

- If even this node is insufficient, it searches for the remote node and performs it.


slowpath-3
kmem_cache_cpu
freelist
cpu_slab
page
kmem_cache
partial
c→partial 마저 비어 있는 경우
kmem_cache_cpu
freelist
cpu_slab
page
ALLOC_FROM_PARTIAL++
slub(frozen)
CPU_PARTIAL_NODE++
partial
slub(frozen)
slub(frozen)
kmem_cache
.cpu_partial=16
n->partial에서 첫 slub을 c→page로 올리고 나머지 일부를 c→partial로 옮긴다.
__slab_alloc() - new_slab: → new_slab_objects() → get_partial()
available=9
s→cpu_partial(캐시가 유지할 object 수)의 절반을 초과할 만큼
node[]
partial
slub
slub
slub
slub
slub
kmem_cache_node
해당 노드에 없으면 리모트 노드에서라도 제한된 비율 이내에서 할당을 시도한다.
get_any_partial()
.remote_node_defrag_ratio

(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab_alloc_node-4f.png)

The following diagram shows the SlowPath 4-step process of moving a slap page from the Buddy system to C->Page & C->Freelist if there is a shortage of slab pages.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab_alloc_node-5b.png)

SlowPath Step 3 Processing Call

get_partial()

mm/slub.c

```
1  /*
2   * Get a partial page, lock it and return it.
3   */
```

```
01 static void *get_partial(struct kmem_cache *s, gfp_t flags, int node,
02                 struct kmem_cache_cpu *c)
03 {
04         void *object;
05         int searchnode = node;
06 
07         if (node == NUMA_NO_NODE)
08                 searchnode = numa_mem_id();
09         else if (!node_present_pages(node))
10                 searchnode = node_to_mem_node(node);
11 
12         object = get_partial_node(s, get_node(s, searchnode), c, flags);
13         if (object || node != NUMA_NO_NODE)
14                 return object;
15 
16         return get_any_partial(s, flags, c);
17 }
```

Move the regular slab pages of the n->partial list to the c->freelist, c->page, and c->partial lists. If the partial list of specified nodes is empty, try on the remote nodes. (Slowpath Step 3)

- Migrate the slab pages from the partial list of nodes specified in code lines 7~14 (adjacent nodes in the case of memoryless nodes) to page and partial in the per CPU cache. If there is a free object to be allocated normally, it returns the object.
- In line 16 of code, we perform the same processing as above with remote nodes.

Slowpath Step 3 – Processing on the Request Node

get_partial_node()

mm/slub.c

```
1  /*
2   * Try to allocate a partial slab from a specific node.
3   */
```

```
01 static void *get_partial_node(struct kmem_cache *s, struct kmem_cache_node *n,
02                                 struct kmem_cache_cpu *c, gfp_t flags)
03 {
04         struct page *page, *page2;
05         void *object = NULL;
06         int available = 0;
07         int objects;
08 
09         /*
10          * Racy check. If we mistakenly see no partial slabs then we
11          * just allocate an empty slab. If we mistakenly try to get a
12          * partial slab and there is none available then get_partials()
13          * will return NULL.
14          */
15         if (!n || !n->nr_partial)
16                 return NULL;
17 
18         spin_lock(&n->list_lock);
19         list_for_each_entry_safe(page, page2, &n->partial, lru) {
20                 void *t;
21 
22                 if (!pfmemalloc_match(page, flags))
23                         continue;
24 
25                 t = acquire_slab(s, n, page, object == NULL, &objects);
26                 if (!t)
27                         break;
28 
29                 available += objects;
30                 if (!object) {
31                         c->page = page;
32                         stat(s, ALLOC_FROM_PARTIAL);
33                         object = t;
34                 } else {
35                         put_cpu_partial(s, page, 0);
36                         stat(s, CPU_PARTIAL_NODE);
37                 }
38                 if (!kmem_cache_has_cpu_partial(s)
39                         || available > s->cpu_partial / 2)
40                         break;
41 
42         }
43         spin_unlock(&n->list_lock);
44         return object;
45 }
```

It retrieves a certain amount of slap pages from a partial list of per nodes and migrates them to the per CPU cache. Returns null with no slab pages to migrate. (Slowpath Step 3 – Local Node)

- In line 15~16 of the code, if the number of slab pages in the per node reaches zero, it returns null.
    - Since there is no node lock, it can be zero in a node lock competition.

- With the node lock obtained in line 18~23 of code, it traverses the slab pages in the partial list of requested nodes, and pfmemalloc skips the unmatched slab pages.
- If the first slap page is fetched in lines 25~33, move it to page in the per CPU cache, and increment the ALLOC_FROM_PARTIAL counter.
- Starting from the second slap page in lines 34~37 of code, move it to the partial list of per cpu cache, and increment the CPU_PARTIAL_NODE counter.
    - put_cpu_partial() function
        - See also: Slub Memory Allocator -7- (Object Unlocked) (http://jake.dothome.co.kr/slub-object-free) | Qc
- In lines 38~40 of code, if the per-cpu exceeds half of the S->cpu_partial value for the number of objects to manage, it exits the loop.
- Returns the free object of the first page moved from line 44 of code.

acquire_slab()

mm/slub.c

```
/*
 * Remove slab from the partial list, freeze it and
 * return the pointer to the freelist.
 *
 * Returns a list of objects or NULL if it fails.
 */
```

```
01  static inline void *acquire_slab(struct kmem_cache *s,
02                  struct kmem_cache_node *n, struct page *page,
03                  int mode, int *objects)
04  {
05          void *freelist;
06          unsigned long counters;
07          struct page new;
08  
09          lockdep_assert_held(&n->list_lock);
10  
11          /*
12           * Zap the freelist and set the frozen bit.
13           * The old freelist is the list of objects for the
14           * per cpu allocation list.
15           */
16          freelist = page->freelist;
17          counters = page->counters;
18          new.counters = counters;
19          *objects = new.objects - new.inuse;
20          if (mode) {
21                  new.inuse = page->objects;
22                  new.freelist = NULL;
23          } else {
24                  new.freelist = freelist;
25          }
26  
27          VM_BUG_ON(new.frozen);
28          new.frozen = 1;
29  
30          if (!__cmpxchg_double_slab(s, page,
31                          freelist, counters,
32                          new.freelist, new.counters,
33                          "acquire_slab"))
34                  return NULL;
35  
```

```
        remove_partial(n, page);
        WARN_ON(!freelist);
        return freelist;
}
```

In order to move one slap page from the partial list of per nodes to the CPU cache, it is detached by making it frozen. The value returned is the first free object, and if it fails, it returns null. According to the @mode, it works as follows:

- If @mode is true, the obtained slab page is obtained for the purpose of moving it to c->page. In order to make all the free objects on the slab page also in use, we substitute page->inuse with page->objects, and specify null for page->freelist.
- If the @mode is false, the obtained slab page is obtained for the purpose of moving it to a c->partial list.

Slowpath Step 3 – Processing on Remote Nodes

get_any_partial()

mm/slub.c

```
/*
 * Get a page from somewhere. Search in increasing NUMA distances.
 */
static void *get_any_partial(struct kmem_cache *s, gfp_t flags,
                struct kmem_cache_cpu *c)
{
#ifdef CONFIG_NUMA
        struct zonelist *zonelist;
        struct zoneref *z;
        struct zone *zone;
        enum zone_type high_zoneidx = gfp_zone(flags);
        void *object;
        unsigned int cpuset_mems_cookie;

        /*
         * The defrag ratio allows a configuration of the tradeoffs between
         * inter node defragmentation and node local allocations. A lower
         * defrag_ratio increases the tendency to do local allocations
         * instead of attempting to obtain partial slabs from other nodes.
         *
         * If the defrag_ratio is set to 0 then kmalloc() always
         * returns node local objects. If the ratio is higher then kmalloc()
         * may return off node objects because partial slabs are obtained
         * from other nodes and filled up.
         *
         * If /sys/kernel/slab/xx/remote_node_defrag_ratio is set to 100
         * (which makes defrag_ratio = 1000) then every (well almost)
         * allocation will first attempt to defrag slab caches on other nodes.
         * This means scanning over all nodes to look for partial slabs which
         * may be expensive if we do it every time we are trying to find a slab
         * with available objects.
         */
33          if (!s->remote_node_defrag_ratio ||
34                          get_cycles() % 1024 > s->remote_node_defrag_ratio)
35                  return NULL;
36
37          do {
38                  cpuset_mems_cookie = read_mems_allowed_begin();
39                  zonelist = node_zonelist(mempolicy_slab_node(), flags);
40                  for_each_zone_zonelist(zone, z, zonelist, high_zoneidx) {
41                          struct kmem_cache_node *n;
42
43                          n = get_node(s, zone_to_nid(zone));
44
45                          if (n && cpuset_zone_allowed(zone, flags) &&
46                                          n->nr_partial > s->min_partial) {
47                                  object = get_partial_node(s, n, c, flags);
48                                  if (object) {
49                                          /*
50                                           * Don't check read_mems_allowed_retry()
51                                           * here - if mems_allowed was updated in
52                                           * parallel, that was a harmless race
53                                           * between allocation and the cpuset
54                                           * update
55                                           */
56                                          return object;
57                                  }
58                          }
59                  }
60          } while (read_mems_allowed_retry(cpuset_mems_cookie));
61  #endif
62          return NULL;
63  }
```

Move the slab page from the partial list of remote nodes to the page and partial list of the per cpu cache in the cpu cache.

- On lines 39~42 of code, use the delay timer to perform the routine at the ratio of "/sys/kernel/slab/<slab >/remote_node_defrag_ratio=" (default=1000(99.8%), 1024=100%). If it is excluded from the probability, it returns null.
    - Delay Loop Timer (http://jake.dothome.co.kr/delay) | 문c
- In line 44~45 of the code, if there is a cpuset fluctuation, the cpuset sequence lock value is read **for retry**
- In line 46~47 of the code, traverse the zones of high_zoneidx or less from top to bottom in the zonelist of nodes found by mempolicy.
- In line 50~65 of the code, only if the node in the traversing zone is a node allowed by CPUset and the number of partial lists of nodes is greater than the minimum S->min_partial required by the cache, a certain amount of slab pages from the partial list of that node will be moved to the page and partial list of the per CPU cache. If successful, it returns the first free object.
- If there is a cpuset churn in line 67 of code, retry
- This is the case when the code is not processed to traverse the entire zonelist. Returns null as a failure.

Atomic replacement of 2 values in slab

Architectures that provide an atomic API that allows you to change 86 (double) values at the same time on x64, arm390, s2, etc., will be handled with slightly faster performance, while those that don't will be replaced with interrupts blocked.

There are two related APIs:

- this_cpu_cmpxchg_double()
  - It is used to replace two values related to the per-cpu variable.
- cmpxchg_double()
  - It is used to replace two values related to a global variable.

Functions that atomically swap 2 values in a slab

- cmpxchg_double_slab()
  - If the architecture doesn't support an API that atomically swaps two values, disable interrupts in the function and handle them with bit_spin_lock.
- __cmpxchg_double_slab()
  - If the architecture doesn't support an API that atomically swaps two values, it will be handled with a bit_spin_lock in the function.
  - The interrupt must be disabled before the function is called.

__cmpxchg_double_slab()

mm/slub.c

```
/* Interrupts must be disabled (for the fallback code to work right) */
static inline bool __cmpxchg_double_slab(struct kmem_cache *s, struct page *page,
                void *freelist_old, unsigned long counters_old,
                void *freelist_new, unsigned long counters_new,
                const char *n)
{
        VM_BUG_ON(!irqs_disabled());
#if defined(CONFIG_HAVE_CMPXCHG_DOUBLE) && \
    defined(CONFIG_HAVE_ALIGNED_STRUCT_PAGE)
        if (s->flags & __CMPXCHG_DOUBLE) {
                if (cmpxchg_double(&page->freelist, &page->counters,
                                   freelist_old, counters_old,
                                   freelist_new, counters_new))
                        return 1;
        } else
#endif
        {
                slab_lock(page);
                if (page->freelist == freelist_old &&
                                        page->counters == counters_old) {
                        page->freelist = freelist_new;
                        page->counters = counters_new;
                        slab_unlock(page);
                        return 1;
                }
                slab_unlock(page);
        }

        cpu_relax();
        stat(s, CMPXCHG_DOUBLE_FAIL);

#ifdef SLUB_DEBUG_CMPXCHG
        pr_info("%s %s: cmpxchg double redo ", n, s->name);
#endif

        return 0;
}
```

Compare page->freelist and counter to the old value and replace it with a new value if it is the same. In architectures that support CONFIG_HAVE_CMPXCHG_DOUBLE, the replacement of two word values is handled atomically. For unsupported architectures, use bit_spin_lock() to handle them.

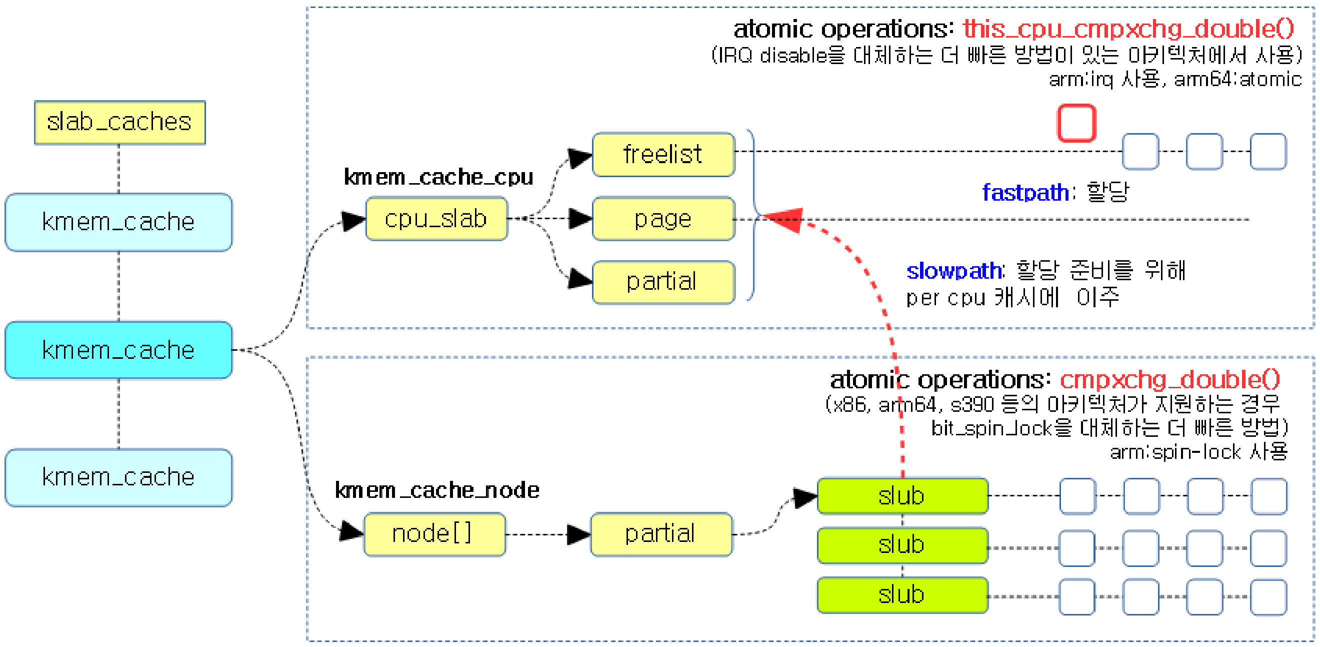
- if @page->freelist == @freelist_old && @page->counters == @counters_old
  - @page->freelist   <— @freelist_new
  - @page->counters <— @counters_new

CONFIG_HAVE_CMPXCHG_DOUBLE

This is a kernel option used by architectures that support atomic APIs that replace 2 (double) values.

- x86, ARM64, and S390 architectures are supported.

The following figure shows two atomic operations: the fastpath used by the per CPU cache and the slowpath used by the per node.


atomic operations: this_cpu_cmpxchg_double()
(IRQ disable을 대체하는 더 빠른 방법이 있는 아키텍처에서 사용)
arm:irq 사용, arm64:atomic
slab_caches
kmem_cache
kmem_cache
kmem_cache
kmem_cache_cpu
cpu_slab
freelist
page
partial
fastpath: 할당
slowpath: 할당 준비를 위해
per cpu 캐시에 이주
atomic operations: cmpxchg_double()
(x86, arm64, s390 등의 아키텍처가 지원하는 경우
bit_spin_lock을 대체하는 더 빠른 방법)
arm:spin-lock 사용
kmem_cache_node
node[]
partial
slub
slub
slub

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

Here's how to atomically swap two values at once:

- Generic method of replacing two values after blocking an interrupt
- For fast performance, the architecture supports atomic API methods – such as the ARM64 architecture.

this_cpu_cmpxchg_double()

include/linux/percpu-defs.h

```
#define this_cpu_cmpxchg_double(pcp1, pcp2, oval1, oval2, nval1, nval2)
\
        __pcpu_double_call_return_bool(this_cpu_cmpxchg_double_, pcp1, pcp2, oval1, oval2, nval1, nval2)
```

If the values of the two per-cpu variables pcp1 and pcp2 are the same as oval1 and oval2, change them to nval1 and nval2.

- if pcp1 == oval1 && pcp2 == oval2
  - pcp1 = nval1
  - pcp2 = nval2

__pcpu_double_call_return_bool()

include/linux/percpu-defs.h

```
/*
 * Special handling for cmpxchg_double.  cmpxchg_double is passed two
 * percpu variables.  The first has to be aligned to a double word
 * boundary and the second has to follow directly thereafter.
 * We enforce this on all architectures even if they don't support
 * a double cmpxchg instruction, since it's a cheap requirement, and it
 * avoids breaking the requirement for architectures with the instruction.
 */
```

```
#define __pcpu_double_call_return_bool(stem, pcp1, pcp2, ...)
\
({
\
        bool pdcrb_ret__;
\
        __verify_pcpu_ptr(&(pcp1));
\
        BUILD_BUG_ON(sizeof(pcp1) != sizeof(pcp2));
\
        VM_BUG_ON((unsigned long)(&(pcp1)) % (2 * sizeof(pcp1)));
\
        VM_BUG_ON((unsigned long)(&(pcp2)) !=
\
                  (unsigned long)(&(pcp1)) + sizeof(pcp1));
\
        switch(sizeof(pcp1)) {
\
        case 1: pdcrb_ret__ = stem##1(pcp1, pcp2, __VA_ARGS__); break;
\
        case 2: pdcrb_ret__ = stem##2(pcp1, pcp2, __VA_ARGS__); break;
\
        case 4: pdcrb_ret__ = stem##4(pcp1, pcp2, __VA_ARGS__); break;
\
        case 8: pdcrb_ret__ = stem##8(pcp1, pcp2, __VA_ARGS__); break;
\
        default:
\
                __bad_size_call_parameter(); break;
\
        }
\
        pdcrb_ret__;
\
})
```

For cmpxchg_double handling, call the function with a suffix of 1, 2, 4, or 8 to the function name that starts with @stem.

- e.g. @stem=this_cpu_cmpxchg_double_
  - this_cpu_cmpxchg_double_1(pcp1, pcp2, …)
  - this_cpu_cmpxchg_double_2(pcp1, pcp2, …)
  - this_cpu_cmpxchg_double_3(pcp1, pcp2, …)
  - this_cpu_cmpxchg_double_4(pcp1, pcp2, …)

this_cpu_cmpxchg_double_8()

include/asm-generic/percpu.h

```
#ifndef this_cpu_cmpxchg_double_8
#define this_cpu_cmpxchg_double_8(pcp1, pcp2, oval1, oval2, nval1, nval2) \
        this_cpu_generic_cmpxchg_double(pcp1, pcp2, oval1, oval2, nval1, nval2)
```

Generic method using interrupt blocking

this_cpu_generic_cmpxchg_double() – generic

include/asm-generic/percpu.h

```
#define this_cpu_generic_cmpxchg_double(pcp1, pcp2, oval1, oval2, nval1, nval2) \
({
\
        int __ret;
\
        unsigned long __flags;
\
        raw_local_irq_save(__flags);
\
        __ret = raw_cpu_generic_cmpxchg_double(pcp1, pcp2,
\
                        oval1, oval2, nval1, nval2);
\
        raw_local_irq_restore(__flags);
\
        __ret;
\
})
```

After masking the interrupt with a generic method, it compares the values of the two per-cpu variables and changes them to the new values.

raw_cpu_generic_cmpxchg_double()

include/asm-generic/percpu.h

```
#define raw_cpu_generic_cmpxchg_double(pcp1, pcp2, oval1, oval2, nval1, nval2) \
({                                                                      \
        typeof(&(pcp1)) __p1 = raw_cpu_ptr(&(pcp1));                    \
        typeof(&(pcp2)) __p2 = raw_cpu_ptr(&(pcp2));                    \
        int __ret = 0;                                                  \
        if (*__p1 == (oval1) && *__p2  == (oval2)) {                    \
                *__p1 = nval1;                                          \
                *__p2 = nval2;                                          \
                __ret = 1;                                              \
        }                                                               \
        (__ret);                                                        \
})
```

Compare the values of the two per-cpu variables and change them to the new values.

Atomic API methods supported by the architecture

this_cpu_generic_cmpxchg_double() – ARM64

arch/arm64/include/asm/percpu.h

```
/*
 * It would be nice to avoid the conditional call into the scheduler when
 * re-enabling preemption for preemptible kernels, but doing that in a way
 * which builds inside a module would mean messing directly with the preempt
 * count. If you do this, peterz and tglx will hunt you down.
 */
```

```
#define this_cpu_cmpxchg_double_8(ptr1, ptr2, o1, o2, n1, n2)          \
({                                                                      \
        int __ret;                                                      \
        preempt_disable_notrace();                                      \
        __ret = cmpxchg_double_local(   raw_cpu_ptr(&(ptr1)),           \
                                        raw_cpu_ptr(&(ptr2)),           \
                                                o1, o2, n1, n2);
\
        preempt_enable_notrace();
\
        __ret;
\
})
```

After masking the preemption using the atomic instructions supported by the architecture, the values of the two per-cpu variables are compared and changed to the new values.

cmpxchg_double_local() – ARM64

arch/arm64/include/asm/cmpxchg.h

```
#define cmpxchg_double_local(ptr1, ptr2, o1, o2, n1, n2) \
({\
        int __ret;\
        __cmpxchg_double_check(ptr1, ptr2); \
        __ret = !__cmpxchg_double((unsigned long)(o1), (unsigned long)(o
2), \
                                  (unsigned long)(n1), (unsigned long)(n
2), \
                                  ptr1); \
        __ret; \
})
```

Use the atomic command, which is supported by the ARM64 architecture, to compare two unsigned long values and change them to the new values.

__cmpxchg_double() Series – ARM64 – ll_sc Method

include/asm/atomic_ll_sc.h

```
#define __CMPXCHG_DBL(name, mb, rel, cl)
\
__LL_SC_INLINE long
\
__LL_SC_PREFIX(__cmpxchg_double##name(unsigned long old1,
\
                                      unsigned long old2,
\
                                      unsigned long new1,
\
                                      unsigned long new2,
\
                                      volatile void *ptr))
\
{
\
        unsigned long tmp, ret;
\
                                                                     \
        asm volatile("// __cmpxchg_double" #name "\n"
\
        "       prfm    pstl1strm, %2\n"
\
        "1:     ldxp    %0, %1, %2\n"
\
        "       eor     %0, %0, %3\n"
\
        "       eor     %1, %1, %4\n"                               \
        "       orr     %1, %0, %1\n"                               \
        "       cbnz    %1, 2f\n"                                   \
        "       st" #rel "xp    %w0, %5, %6, %2\n"                  \
        "       cbnz    %w0, 1b\n"                                  \
        "       " #mb "\n"                                          \
        "2:"                                                        \
        : "=&r" (tmp), "=&r" (ret), "+Q" (*(unsigned long *)ptr)    \
        : "r" (old1), "r" (old2), "r" (new1), "r" (new2)            \
        : cl);                                                      \
                                                                    \
        return ret;                                                 \
}                                                                   \
__LL_SC_EXPORT(__cmpxchg_double##name);

__CMPXCHG_DBL(   ,        ,  ,         )
__CMPXCHG_DBL(_mb, dmb ish, l, "memory")
```

The ARM64 architecture uses ll_sc atomic commands to compare two unsigned long values and change them to the new values.

__cmpxchg_double() Series – ARM64 – LSE Method

```
#define __LL_SC_CMPXCHG_DBL(op) __LL_SC_CALL(__cmpxchg_double##op)

#define __CMPXCHG_DBL(name, mb, cl...)                              \
static inline long __cmpxchg_double##name(unsigned long old1,       \
                                          unsigned long old2,       \
                                          unsigned long new1,       \
                                          unsigned long new2,       \
                                          volatile void *ptr)       \
{                                                                   \
        unsigned long oldval1 = old1;                               \
        unsigned long oldval2 = old2;                               \
        register unsigned long x0 asm ("x0") = old1;                \
        register unsigned long x1 asm ("x1") = old2;                \
        register unsigned long x2 asm ("x2") = new1;                \
        register unsigned long x3 asm ("x3") = new2;                \
        register unsigned long x4 asm ("x4") = (unsigned long)ptr;  \
                                                                                \
        asm volatile(ARM64_LSE_ATOMIC_INSN(                                     \
        /* LL/SC */                                                             \
        __LL_SC_CMPXCHG_DBL(name)                                               \
        __nops(3),                                                              \
        /* LSE atomics */                                                       \
        "       casp" #mb "\t%[old1], %[old2], %[new1], %[new2], %[v]\n"\
        "       eor     %[old1], %[old1], %[oldval1]\n"                         \
        "       eor     %[old2], %[old2], %[oldval2]\n"                         \
        "       orr     %[old1], %[old1], %[old2]")                             \
        : [old1] "+&r" (x0), [old2] "+&r" (x1),                                 \
          [v] "+Q" (*(unsigned long *)ptr)                                      \
        : [new1] "r" (x2), [new2] "r" (x3), [ptr] "r" (x4),                     \
          [oldval1] "r" (oldval1), [oldval2] "r" (oldval2)                      \
        : __LL_SC_CLOBBERS, ##cl);                                              \
                                                                                \
        return x0;                                                              \
}

__CMPXCHG_DBL(   ,   )
__CMPXCHG_DBL(_mb, al, "memory")
```

Using the lse-like atomic command, which is supported by the ARM64 architecture, two unsigned long values are compared and changed to the new values.

cmpxchg_double() – generic

include/asm-generic/atomic-instrumented.h

```
#define cmpxchg_double(p1, p2, o1, o2, n1, n2)                                  \
({                                                                              \
        typeof(p1) __ai_p1 = (p1);                                              \
        kasan_check_write(__ai_p1, 2 * sizeof(*__ai_p1));                       \
        arch_cmpxchg_double(__ai_p1, (p2), (o1), (o2), (n1), (n2));             \
})
```

cmpxchg_double() – ARM64

arch/arm64/include/asm/cmpxchg.h

```
#define cmpxchg_double(ptr1, ptr2, o1, o2, n1, n2) \
({\
        int __ret;\
        __cmpxchg_double_check(ptr1, ptr2); \
        __ret = !__cmpxchg_double_mb((unsigned long)(o1), (unsigned long)(o2), \
                                     (unsigned long)(n1), (unsigned long)(n2), \
                                     ptr1); \
        __ret; \
})
```

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) | Sentence C – Current post
- 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)) | 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

---

12 thoughts to “Slub Memory Allocator -6- (Object 할당)”

HYEYOO (HTTPS://HYEYOO.COM)
2021-09-18 02:26 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306028)

Hello, I have a question while studying, so I leave a comment. Slowpath – We have a function get_freelist from 1, and in this inline function we change the value of new, where new is a local variable.

new.inuse = page->objects;
new.frozen = freelist != NULL;

So why change the structure new? Is this accessible from outside the function get_freelist?
I think I've missed either the C language or the race condition.

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306028#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-09-18 11:34 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306030)

Hello?

The local variable new, which is created with a page structure, is being used temporarily, as you said.
However, new.counter is declared as a union, so if you check it, you'll understand it quickly.
new.counter = new.inuse | new.objects | new.frozen

```
 참고) include/linux/mm_types.h
union {
void *s_mem; /* slab: first object */
unsigned long counters; /* SLUB */
struct { /* SLUB */
unsigned inuse:16;
unsigned objects:15;
unsigned frozen:1;
};
};
```

I appreciate it.

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306030#RESPOND)

HYEYOO (HTTPS://NAVER.COM-)
2021-09-18 12:01 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306032)

Thank you for your reply. But the gist of my question was what is the point of storing a value in new, a temporary variable used by get_freelist,
and then not using it at all?

The function looks like this, but I didn't use it at all after changing new.
static inline void *get_freelist(struct kmem_cache *s, struct page *page)
{
struct page new;
unsigned long counters;
void *freelist;

```
do {
freelist = page->freelist;
counters = page->counters;

new.counters = counters;
VM_BUG_ON(!new.frozen);

new.inuse = page->objects;
new.frozen = freelist != NULL;

} while (!__cmpxchg_double_slab(s, page,
freelist, counters,
NULL, new.counters,
“get_freelist”));

return freelist;
}
```

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306032#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-09-18 14:35 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306033)

The cmpxchg_double_slab() command compares two values with two old values, and then substitutes (replaces) two new values only if they are the same.
In the end, if
the values of page->freelist and page->counters are the same compared to freelist and counters, then page->freelist = NULL, and page->counters = new.counters values.

So we used the changed value of new. (new.inuse and new.frozen are reflected in new.counters)
We didn't use all of the new structs, but we did use the new.counters section.

I appreciate it.

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306033#RESPOND)

HYEYOO (HTTPS://HYEYOO.COM)
2021-09-18 16:47 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306034)

Oh boy...!! I forgot that counters are a union of inuse/frozen/objects.
Not for bitwise, but for the use of unions! I appreciate it.

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306034#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-09-18 18:58 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306035)

Yes, fortunately, you found out right away. Have a nice day. ^^

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306035#RESPOND)

HYEYOO (HTTPS://HYEYOO.COM)
2021-09-18 02:31 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306029)

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cc09ee80c3b1 (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cc09ee80c3b1)
with the recent merge of PREEMPT_RT, slap and so on have changed a lot

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306029#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-09-18 11:43 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306031)

Thanks for the great information. I'll have to read it carefully.

Although PREEMPT_RT has been merged into the current kernel, some of the works that have been done on the actual RT kernel have not yet been migrated for stability and compatibility with existing kernels.
Thus, CONFIG_PREMPT_RT kernel options are slightly more preemtinable than CONFIG_PREEMPT when enabled by individuals, but they are a little different from the previous RT kernels.
This shortfall requires SoC vendors to do more work to release it.

I appreciate it.

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306031#RESPOND)

AS IT IS
2021-11-06 19:40 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306108)

Hello

I'm asking because I don't understand the get_freelist() function.

I compared freelist and page->freelist, counters and page->counters in do-while statements, and understood them to be syntax that exits the while statement if they are the same. One thing I'm curious about is when it happens that the two conditions are not the same. There is no syntax for re-assigning the values of the freelist and counters variables, and there doesn't seem to be any change to the page structure, so what are the cases where the two conditions are not the same?

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306108#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-11-08 10:39 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306111)

Hello?

In SMP systems, page->freelist can be contested when another cpu frees the page's slab object via the __slab_free() function.

I appreciate it.

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306111#RESPOND)

AS IT IS
2021-11-08 21:32 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306112)

Oh I see!

I've been looking at the SLUB allocator in my study lately, and I think the material you've put together really helps me a lot.
Thank you as always!

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306112#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-11-09 09:01 (http://jake.dothome.co.kr/slub-object-alloc/#comment-306116)

I hope this helps you analyze. Have a nice day!

RESPONSE (/SLUB-OBJECT-ALLOC/?REPLYTOCOM=306116#RESPOND)

LEAVE A COMMENT

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

Comments

name *

email *

Website

WRITE A COMMENT

Slub Memory Allocator -8- (Drain/Flush Cache) (http://jake.dothome.co.kr/slub-drain-flush-cache/)

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