# Slub Memory Allocator -8- (Drain/Flush Cache)

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

<kernel v5.0>

## Drain Cache

### per-cpu slab cache page -> n->partial Go to list

#### deactivate_slab()

mm/slub.c -1/3-

```
/*
 * Remove the cpu slab
 */
```

```
static void deactivate_slab(struct kmem_cache *s, struct page *page,
                                void *freelist, struct kmem_cache_cpu *c)
{
        enum slab_modes { M_NONE, M_PARTIAL, M_FULL, M_FREE };
        struct kmem_cache_node *n = get_node(s, page_to_nid(page));
        int lock = 0;
        enum slab_modes l = M_NONE, m = M_NONE;
        void *nextfree;
        int tail = DEACTIVATE_TO_HEAD;
        struct page new;
        struct page old;

        if (page->freelist) {
                stat(s, DEACTIVATE_REMOTE_FREES);
                tail = DEACTIVATE_TO_TAIL;
        }

        /*
         * Stage one: Free all available per cpu objects back
         * to the page freelist while it is still frozen. Leave the
         * last one.
         *
         * There is no need to take the list->lock because the page
         * is still frozen.
         */
        while (freelist && (nextfree = get_freepointer(s, freelist))) {
                void *prior;
                unsigned long counters;

                do {
                        prior = page->freelist;
                        counters = page->counters;
                        set_freepointer(s, freelist, prior);
                        new.counters = counters;
                        new.inuse--;
                        VM_BUG_ON(!new.frozen);

                } while (!__cmpxchg_double_slab(s, page,
                        prior, counters,
```

```
40                              freelist, new.counters,
41                              "drain percpu freelist"));
42
43                      freelist = nextfree;
44              }
```

Move the per-cpu slab page to an n->partial list.

- In code lines 13~16, increment the DEACTIVASTE_REMOTE_FREES counter if there are free objects on the page.n->partial to prepare them to be aft when added to the list.
    - If the object is free on the remote CPU, the free object may exist in the page->freelist. After all, the page was processed by the remote CPU, so if possible, it should be added (colded) to the end of the n->partial list to slow down the probability that the current CPU will be able to access it.

**Stage 1 – c->freelist except the last one—> page->freelist**

- In lines 26~44 of the code, except for the last object in the @freelist, it is traversed and moved to page->freelist. Decrements the inuse counter by the number of objects moved.

mm/slub.c -2/3-

```
        /*
         * Stage two: Ensure that the page is unfrozen while the
         * list presence reflects the actual number of objects
         * during unfreeze.
         *
         * We setup the list membership and then perform a cmpxchg
         * with the count. If there is a mismatch then the page
         * is not unfrozen but the page is on the wrong list.
         *
         * Then we restart the process which may have to remove
         * the page from the list that we just put it on again
         * because the number of objects in the slab may have
         * changed.
         */
redo:

        old.freelist = page->freelist;
        old.counters = page->counters;
        VM_BUG_ON(!old.frozen);

        /* Determine target state of the slab */
        new.counters = old.counters;
        if (freelist) {
                new.inuse--;
                set_freepointer(s, freelist, old.freelist);
                new.freelist = freelist;
        } else
                new.freelist = old.freelist;

        new.frozen = 0;

        if (!new.inuse && n->nr_partial >= s->min_partial)
                m = M_FREE;
        else if (new.freelist) {
                m = M_PARTIAL;
                if (!lock) {
                        lock = 1;
                        /*
                         * Taking the spinlock removes the possiblity
                         * that acquire_slab() will see a slab page that
```

```
41 				 * is frozen
42 				 */
43 				spin_lock(&n->list_lock);
44 			}
45 		} else {
46 			m = M_FULL;
47 			if (kmem_cache_debug(s) && !lock) {
48 				lock = 1;
49 				/*
50 				 * This also ensures that the scanning of full
51 				 * slabs from diagnostic functions will not see
52 				 * any frozen slabs.
53 				 */
54 				spin_lock(&n->list_lock);
55 			}
56 		}
```

**Stage 2: Move per-cpu slab pages to the n->partial list.**

When you enter this routine for the first time, there is still one object processing left in the frozen state. By default, the slab page is moved to an n->partial list. However, if all the objects in the slab page are free, it will just be returned to the buddy if debugging is not enabled, and the n->full list will be moved if debugging is enabled.

- On line 15 of the code, the redo: label is. This is where page->freelist comes back if the atomic operation fails.
- In lines 17~18 of the code, back up the freelist and counters (inuse, objects, frozen bits) of the current slab page in the old variable.
- Copy the old counter from code lines 22~30 and prepare the new counter to change. If there is a freelist with one object remaining, it decreases the number of objects used and prepares to insert the last free object in front of the free objects. The slap page is about to be changed to unfronzen status.
- In lines 32~33 of code, if there is no object in use and the n->partial list is in the overflow state, M_FREE the current mode state in order to release the slab page and run it to the buddy system.
- In line 34~44 of code, if there is even one free object, it will be added to the n->partial list, and the current mode state should be M_PARTIAL.
- If there is no free object in line 45~56, it will be added to the n->full list, and the current mode will be M_FULL.

mm/slub.c -3/3-

```
		if (l != m) {
			if (l == M_PARTIAL)
				remove_partial(n, page);
			else if (l == M_FULL)
				remove_full(s, n, page);

			if (m == M_PARTIAL)
				add_partial(n, page, tail);
			else if (m == M_FULL)
				add_full(s, n, page);
		}

		l = m;
		if (!__cmpxchg_double_slab(s, page,
					old.freelist, old.counters,
					new.freelist, new.counters,
```

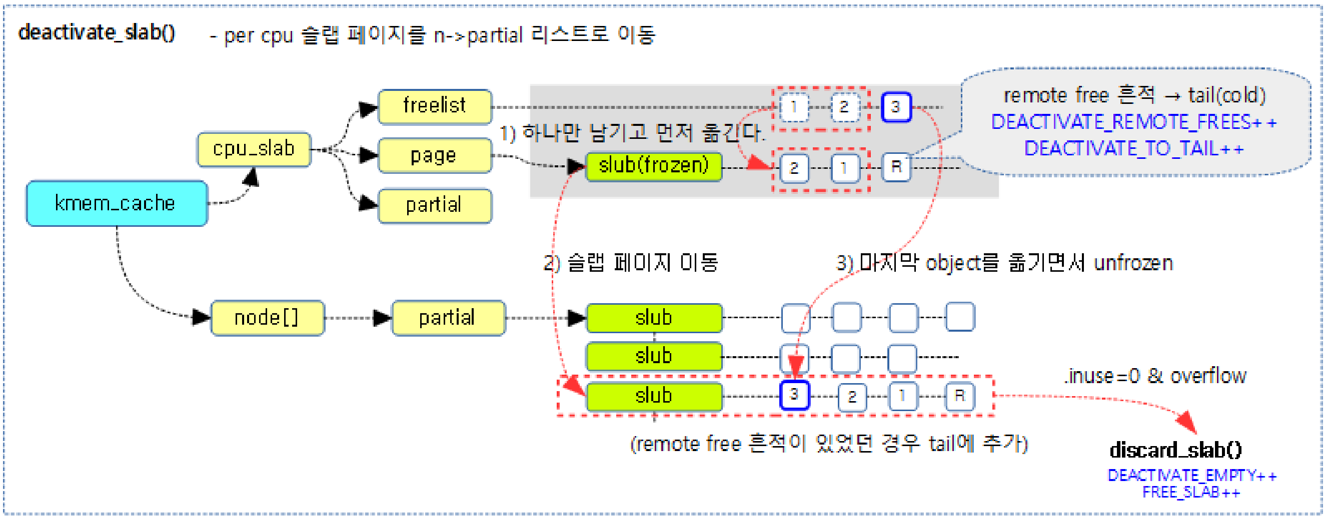
```
17                                 "unfreezing slab"))
18                 goto redo;
19 
20         if (lock)
21                 spin_unlock(&n->list_lock);
22 
23         if (m == M_PARTIAL)
24                 stat(s, tail);
25         else if (m == M_FULL)
26                 stat(s, DEACTIVATE_FULL);
27         else if (m == M_FREE) {
28                 stat(s, DEACTIVATE_EMPTY);
29                 discard_slab(s, page);
30                 stat(s, FREE_SLAB);
31         }
32 
33         c->page = NULL;
34         c->freelist = NULL;
35 }
```

While unfreezing the slab page, the slab page is processed according to the existing mode state l and the current mode state m as follows, but if it fails, it is processed back from the Stage 2 process.

- Add or remove n->partial to the list
- n->full Add or remove to list
- Dismiss the slap page and return it to the buddy system

- In lines 1~11 of the code, the slab page was added to n->partial or n->full, but the state changed, and it was removed again and then re-added to the appropriate location.
  - The allocation and release of free objects to the frozen slab page continues in a contested state.
  - e.g. n->partial, tried atomic processing, but failed. When the slab page is full, it undoes what was added to n->partial and adds it back to n->full.
- In line 13~18 of the code, the mode is the same, and the slab page is handled as follows. If the atomic operation fails, go to the redo: label and try again.
  - if page->freelist == old.freelist & page->counters == old.counters
    - page->freelist = new.freelist
    - page->counters = new.counters
- In line 20~21 of the code, if the lock is applied during that time, release it.
- In line 23~31 of the code, perform the following for the final determined mode.
  - Increments the DEACTIVATE_TO_HEAD or DEACTIVATE_TO_TAIL counters if added to the final n->partial list.
  - Increments the DEACTIVATE_FULL counter if added to the final n->full list.
  - If it is necessary to return to the final buddy system, increment the DEACTIVATE_EMPTY counter and FREE_SLAB counter and send it back to the buddy system.
- In lines 33~34 of code, empty c->page and c->freelist.

The figure below shows the process of moving a per cpu frozen slab page to an n->partial listo.



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

---

# Flush Cache

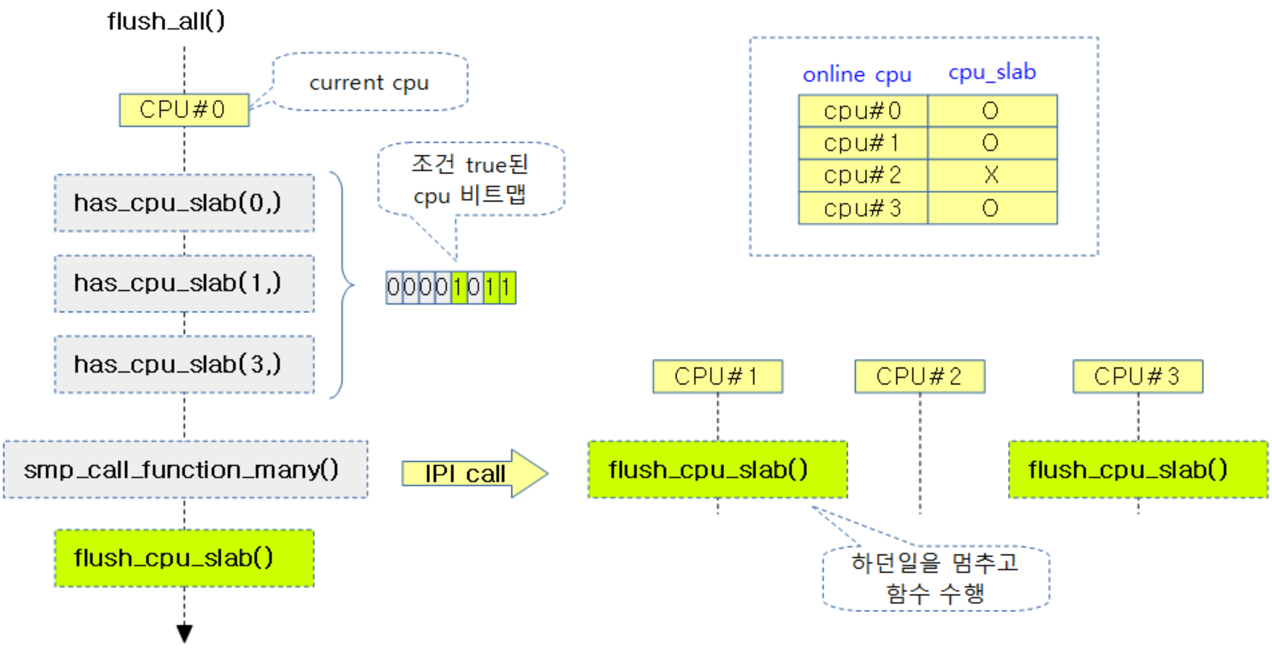
## flush_all()

mm/slub.c

```
static void flush_all(struct kmem_cache *s)
{
        on_each_cpu_cond(has_cpu_slab, flush_cpu_slab, s, 1, GFP_ATOMIC);
}
```

If there are per-CPU slab pages in the slab cache requested by online CPUs, ask each CPU to flush them and move them to the n->partial list, and wait for them to complete.

The figure below shows the process of calling the flush_cpu_slab() function, which flushes the per-cpu slab page of the slab cache if it exists for the four online CPUs.




(http://jake.dothome.co.kr/wp-content/uploads/2016/05/flush_all-1.png)

## has_cpu_slab()

mm/slub.c

```
static bool has_cpu_slab(int cpu, void *info)
{
        struct kmem_cache *s = info;
        struct kmem_cache_cpu *c = per_cpu_ptr(s->cpu_slab, cpu);

        return c->page || c->partial;
}
```

Returns the presence or absence of per-CPU slab pages in the slab cache.

## flush_cpu_slab()

mm/slub.c

```
static void flush_cpu_slab(void *d)
{
        struct kmem_cache *s = d;

        __flush_cpu_slab(s, smp_processor_id());
}
```

Empty the per-CPU slab pages for the current CPU in the slab cache and send them to an n->partial list.

- Move the slab pages of c->page and c->partial lists to n->partial lists.
- This is the function that each CPU's IPI handler is called and executed, and the interrupt must be called with the interrupt disabled.

### __flush_cpu_slab()

mm/slub.c

```
/*
 * Flush cpu slab.
 *
 * Called from IPI handler with interrupts disabled.
 */
```

```
static inline void __flush_cpu_slab(struct kmem_cache *s, int cpu)
{
        struct kmem_cache_cpu *c = per_cpu_ptr(s->cpu_slab, cpu);

        if (likely(c)) {
                if (c->page)
                        flush_slab(s, c);

                unfreeze_partials(s, c);
        }
}
```

Empty the per-cpu slab pages for the @cpu of the slab cache and send them to an n->partial list.

### flush_slab()

mm/slub.c

```
static inline void flush_slab(struct kmem_cache *s, struct kmem_cache_cpu *c)
{
        stat(s, CPUSLAB_FLUSH);
        deactivate_slab(s, c->page, c->freelist, c);

        c->tid = next_tid(c->tid);
}
```

Unfrozen the per-cpu slab page in the slab cache and send it to an n->partial list. and increases CPUSLAB_FLUSH counters.

---

## SMP-related APIs

### on_each_cpu_cond()

kernel/smp.c

```
void on_each_cpu_cond(bool (*cond_func)(int cpu, void *info),
                        smp_call_func_t func, void *info, bool wait,
                        gfp_t gfp_flags)
{
        on_each_cpu_cond_mask(cond_func, func, info, wait, gfp_flags,
                                cpu_online_mask);
}
EXPORT_SYMBOL(on_each_cpu_cond);
```

If a cpu_slab exists on the CPUs that have been online, it asks each CPU to flush the cpu_slab and wait for it to complete.

### on_each_cpu_cond_mask()

kernel/smp.c

```
/*
 * on_each_cpu_cond(): Call a function on each processor for which
 * the supplied function cond_func returns true, optionally waiting
 * for all the required CPUs to finish. This may include the local
 * processor.
 * @cond_func:  A callback function that is passed a cpu id and
 *              the the info parameter. The function is called
 *              with preemption disabled. The function should
 *              return a blooean value indicating whether to IPI
 *              the specified CPU.
 * @func:       The function to run on all applicable CPUs.
 *              This must be fast and non-blocking.
 * @info:       An arbitrary pointer to pass to both functions.
 * @wait:       If true, wait (atomically) until function has
 *              completed on other CPUs.
 * @gfp_flags:  GFP flags to use when allocating the cpumask
 *              used internally by the function.
 *
 * The function might sleep if the GFP flags indicates a non
 * atomic allocation is allowed.
 *
 * Preemption is disabled to protect against CPUs going offline but not online.
 * CPUs going online during the call will not be seen or sent an IPI.
 *
 * You must not call this function with disabled interrupts or
 * from a hardware interrupt handler or from a bottom half handler.
 */
```

```
void on_each_cpu_cond_mask(bool (*cond_func)(int cpu, void *info),
                        smp_call_func_t func, void *info, bool wait,
                        gfp_t gfp_flags, const struct cpumask *mask)
{
        cpumask_var_t cpus;
        int cpu, ret;

        might_sleep_if(gfpflags_allow_blocking(gfp_flags));

        if (likely(zalloc_cpumask_var(&cpus, (gfp_flags|__GFP_NOWARN)))) {
                preempt_disable();
                for_each_cpu(cpu, mask)
                        if (cond_func(cpu, info))
                                __cpumask_set_cpu(cpu, cpus);
                on_each_cpu_mask(cpus, func, info, wait);
                preempt_enable();
                free_cpumask_var(cpus);
        } else {
                /*
                 * No free cpumask, bother. No matter, we'll
                 * just have to IPI them one by one.
                 */
                preempt_disable();
                for_each_cpu(cpu, mask)
                        if (cond_func(cpu, info)) {
                                ret = smp_call_function_single(cpu, func,
                                                                info, wait);
                                WARN_ON_ONCE(ret);
                        }
                preempt_enable();
        }
}
EXPORT_SYMBOL(on_each_cpu_cond_mask);
```

@mask CPUs will perform the @func@cond_func() function if the result of the argument is true. And if the @wait is true, it waits for the completion of the execution request of the function sent by the current CPU to the respective CPU.

- If a __GFP_DIRECT_RECLAIM flag is requested in line 8 of code, perform a preemption pointer.
- Clear the CPU bitmap in line 10~17 of code, loop around the number of CPUs @mask, and set the current CPU bit of the CPU bitmap if the result value is true for the @cond_func() function passed as an argument. It then performs the @func function via an IPI call on any cpu configured in the cpus bitmap. If @wait is true, wait for each function to finish executing.
- In line 18~31 of the code, if no memory is allocated via zalloc_cpumask_var(), it loops around the online cpu and asks the @func function to be executed on the cpu if the result of the @cond_func() function passed as an argument is true.

#### CONFIG_CPUMASK_OFFSTACK Kernel Options

- This is a kernel option that can be used if you are using DEBUG_PER_CPU_MAPS kernel options for CPU debugging.
- This option does not use the CPU mask and dynamically receives a separate memory allocation, so the zalloc_cpumask_var() function does not cause stack overflow, but the disadvantage is that it slows down the speed.

### on_each_cpu_mask()

kernel/smp.c

```
/**
 * on_each_cpu_mask(): Run a function on processors specified by
 * cpumask, which may include the local processor.
 * @mask: The set of cpus to run on (only runs on online subset).
 * @func: The function to run. This must be fast and non-blocking.
 * @info: An arbitrary pointer to pass to the function.
 * @wait: If true, wait (atomically) until function has completed
 *        on other CPUs.
 *
 * If @wait is true, then returns once @func has returned.
 *
 * You must not call this function with disabled interrupts or from a
 * hardware interrupt handler or from a bottom half handler.  The
 * exception is that it may be used during early boot while
 * early_boot_irqs_disabled is set.
 */
```

```
void on_each_cpu_mask(const struct cpumask *mask, smp_call_func_t func,
                        void *info, bool wait)
{
        int cpu = get_cpu();

        smp_call_function_many(mask, func, info, wait);
        if (cpumask_test_cpu(cpu, mask)) {
                unsigned long flags;
                local_irq_save(flags);
                func(info);
                local_irq_restore(flags);
        }
        put_cpu();
}
```

```
15 EXPORT_SYMBOL(on_each_cpu_mask);
```

Perform the @func function on all CPUs configured on the @mask. If @wait is true, wait for each function to finish executing.

- In line 6 of the code, let the @func function run on all remote CPUs except the current CPU. If you specify @wait argument to be true, the current CPU waits for each CPU to finish executing its functions.
  - See: IPI cross call – soft interrupt (http://jake.dothome.co.kr/ipi-cross-call/) | Qc
- In line 7~12 of the code, if the current CPU is also masked, execute the @func function.

## consultation

- Slab Memory Allocator -1- (Structure) (http://jake.dothome.co.kr/slub/) | Qc
- Slab Memory Allocator -2- (Initialize Cache) (http://jake.dothome.co.kr/kmem_cache_init) | Qc
- Slub Memory Allocator -3- (Create Cache) (http://jake.dothome.co.kr/slub-cache-create) | Qc
- Slub Memory Allocator -4- (Calculate Order) (http://jake.dothome.co.kr/slub-order) | Qc
- Slub Memory Allocator -5- | (http://jake.dothome.co.kr/slub-slub-alloc) Qc
- Slub Memory Allocator -6- (Assign Object) (http://jake.dothome.co.kr/slub-object-alloc) | Qc
- Slub Memory Allocator -7- (Object Unlocked) (http://jake.dothome.co.kr/slub-object-free) | Qc
- Slub Memory Allocator -8- (Drain/Flash Cache) (http://jake.dothome.co.kr/slub-drain-flush-cache) | Sentence C – Current post
- 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

---

## 3 thoughts to “Slub Memory Allocator -8- (Drain/Flush 캐시)”

**IPARAN (HTTPS://WWW.BHRAL.COM/)**
2021-12-03 13:19 (http://jake.dothome.co.kr/slub-drain-flush-cache/#comment-306161)

Hello, Moon Young-il~
This is the 16th I.M.Root Iparan ~
I will ask you one more question in a row. ^^;

** The figure below shows the process of moving a per cpu frozen slab page to an n->partial listo.
// deactivate_slab-1a.png

Why do we exclude the last one in drain c->freelist?
Well, what's the purpose of moving except one?

p.s. Oh, and in terms of study, we have about two weeks left in December, except for Christmas and New Year's Day.

I've seen everything from 12 ~ 2 to Slop, and I think I've looked at the memory subsystem + cgroup basics in general over the course of 1 years.

What topics should I start studying? The
items that come to mind as candidates are: (1) Is it better to analyze the code in order after start_kernel -> mm_init -> kem cache init?
(2) Is it better to look at the ARM Linux kernel in order after the slabs in code?
(3) Let's take a look at a specific chapter on understanding the Linux kernel. Process? fork ?
(4) CPU manual?

I've been taking two weeks off in a row since the end of December, so
I'm trying to figure out what to look at before then.

RESPONSE (/SLUB-DRAIN-FLUSH-CACHE/?REPLYTOCOM=306161#RESPOND)

**MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)**
2021-12-05 13:37 (http://jake.dothome.co.kr/slub-drain-flush-cache/#comment-306168)

As we go through the loop, we move the objects from the c->freelist to page->freelist while keeping them frozen, and at the end we move them to a partial list of nodes, and then we process the last 1 object separately in the logic configuration so that we can atomically process the last object and the unfrozen at the same time.

It is processed in the following sequence:
1) Move n-1 objects from c->freelist to page->freelist
2) move the above page to n->partial
3) move the last 1 object from c->freelist to page->freelist and unfreeze at the same time.

—————–

After memory management, the study course suggests the following:
1) Clocks -> counters/timers -> time management -> interrupts/gic -> schedulers -> process/task management (fork)

In the future, you may also be interested in CMA, DMA (Coherent), IOMMU, etc.

I appreciate it.

RESPONSE (/SLUB-DRAIN-FLUSH-CACHE/?REPLYTOCOM=306168#RESPOND)

**IPARAN (HTTPS://WWW.BHRAL.COM/)**

2021-12-05 21:20 (http://jake.dothome.co.kr/slub-drain-flush-cache/#comment-306169)

Oops, it was in the action in Figure 3, but I missed it!

Thank you so much!

p.s
Thanks for the answer! I'm going to share
it with the study members and look at it in the order below regarding the clock based on the current Moon C blog!
– Common Clock Framework -1, 2 – Timer Series

– time_init()

Thank you for the details, as always!

RESPONSE (/SLUB-DRAIN-FLUSH-CACHE/?REPLYTOCOM=306169#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 -12- (debugging slub) (http://jake.dothome.co.kr/slub-debug/)

Slub Memory Allocator -6- (Assign Object) (http://jake.dothome.co.kr/slub-object-alloc/)

Munc Blog (2015 ~ 2024)