Slab Memory Allocator -1- (Structure)

2016-06-03 (http://jake.dothome.co.kr/slub/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Slab Memory Allocator

Slab (Slub, Slob) An object is the smallest unit of regular memory allocation used by the kernel. The kernel is built and used in one of three ways: Learn about the differences between them. When explaining without distinguishing between differences, it is expressed in Hangul as a slap. In addition, both structural and source analysis only analyze the Slub implementation.

Slab

- It was used as the default core of kernel memory management until 2007~8.
- Slab object queues of arrays are used by the CPU and nodes.
- Metadata fragments are placed in front of each other, making it difficult to align objects, which requires very complex processing when cleaning the cache due to lack of memory.
- The first slab created is managed in a full list, but if any object is used, the slab is moved to a partial list and managed. When it runs out, it is moved back to the empty list.
- SLABs are managed on a per-node and per-CPU basis to improve performance.

Slub

- Since 2007~8, it has been used not only in embedded systems with enough memory, but also in PCs and servers, and is currently used as the default.
- Unlike Slab, it simply specifies a single page of slub pages instead of a Slab object queue to reduce memory overhead.
- Unlike Slab, it doesn't use any space for freelist management.
- Unlike Slab, the first slub created starts with 0 objects and is managed as a partial list. If the object is used up, the slub is removed from the partial list and removed from the management mechanism. However, if the slub is free even one object, it is added back to the partial list and managed.
- Slub is also managed by per-node and per-cpu, respectively, to improve performance in the same way as slab.
- Compared to Slab, the use of slub reduced the slab cache in the system (about 50%), increased the locality of the slab allocator, and reduced fragmentation of the slab memory.

Slob

- Choose and use it on embedded Linux with low memory footprint.
- It has the slowest speed but the least memory consumption.

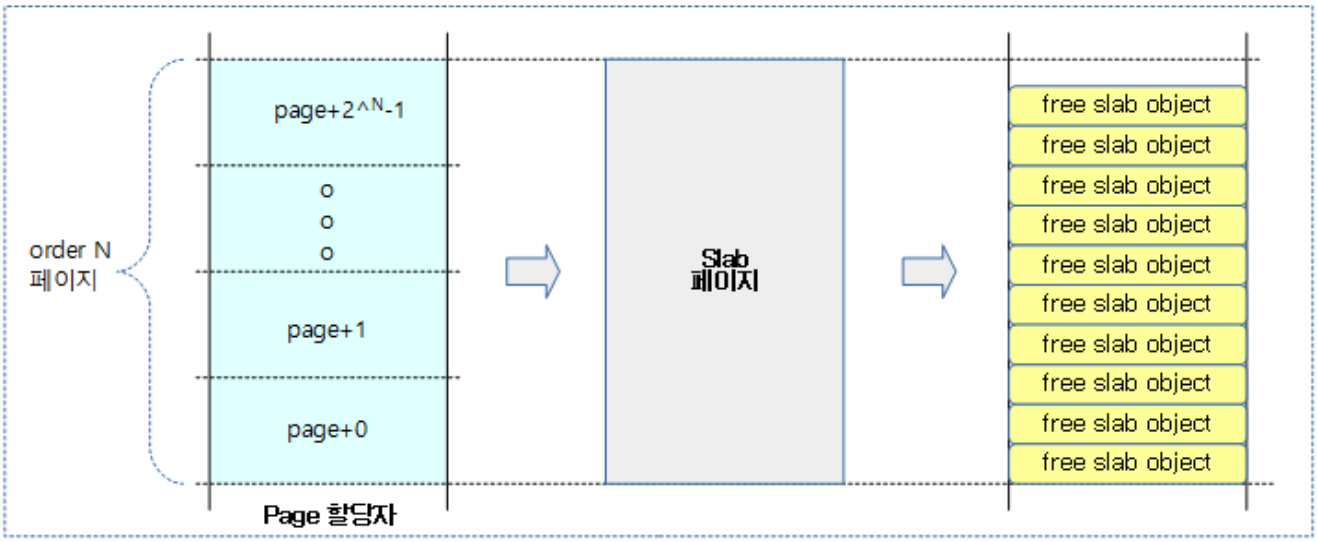
Slab Structure

Slap objects are placed in one or more page frames, and they are used by the specified object size. Meta information may be included in the object for debug information.

Placing an object in a slab page

The order page calculated according to the slab object size is assigned by the buddy system to form one slab page. Fill all slab pages with the same slab object size. The calculated order is used from 1~0 by default.

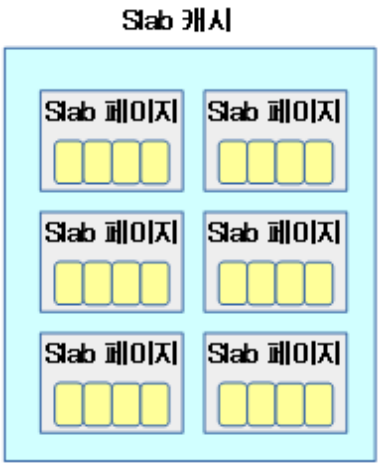
The following figure shows how the buddy system uses order-N pages to construct a slab page, and places assignable slab objects on the slab page.



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

Slap Cache

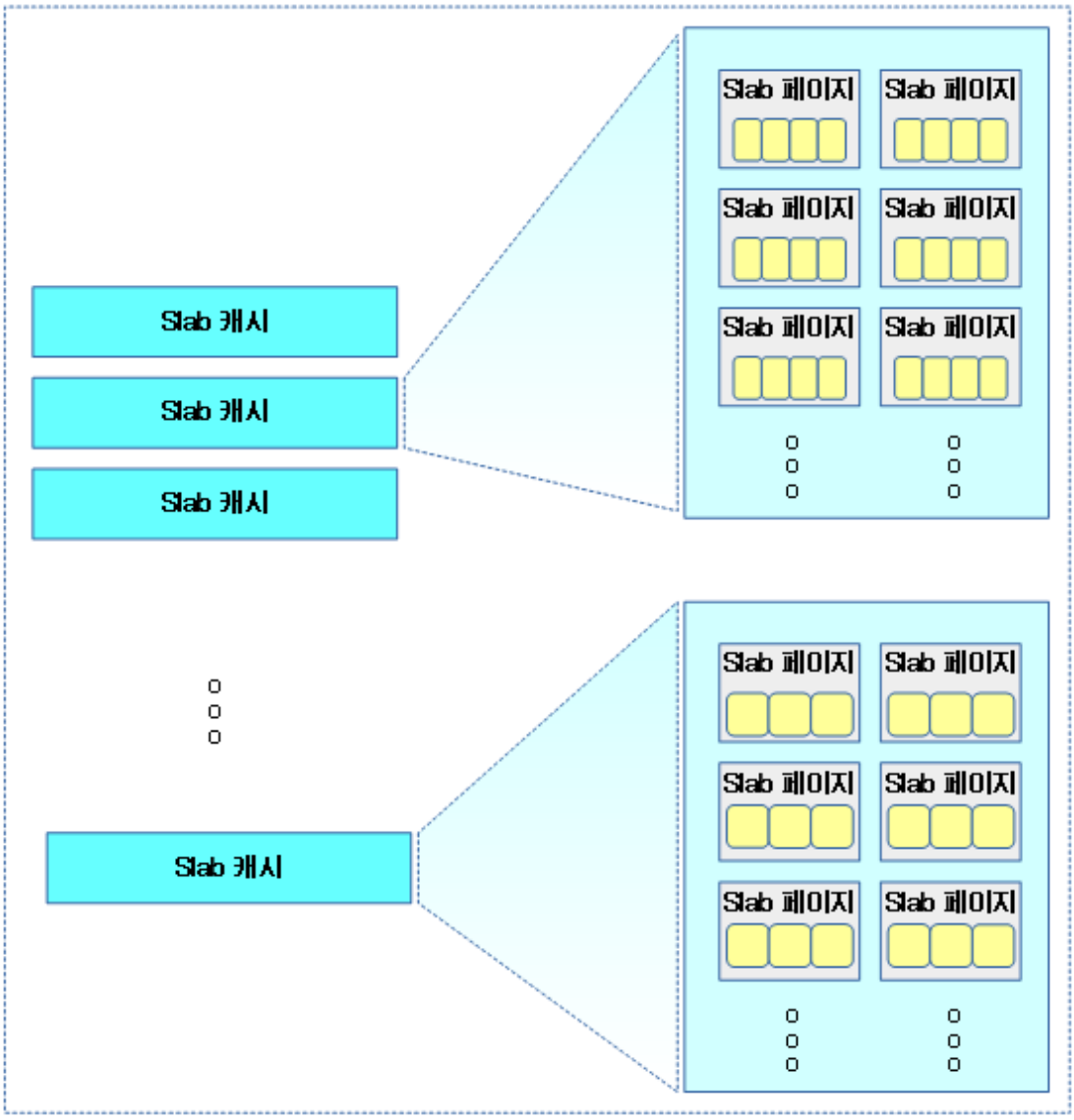
The following figure and one or more slab pages are gathered together to form a slab cache. In other words, all objects in the slab cache provide the same size.



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

As shown in the following figure, if the required object size is different, a slap cache can be created and configured for each object size. If the kernel uses a particular structure a lot, it prepares a slab cache by registering it in advance.

- e.g. page, anon_vma, vm_area_struct, task_struct, dentry, skb, ...



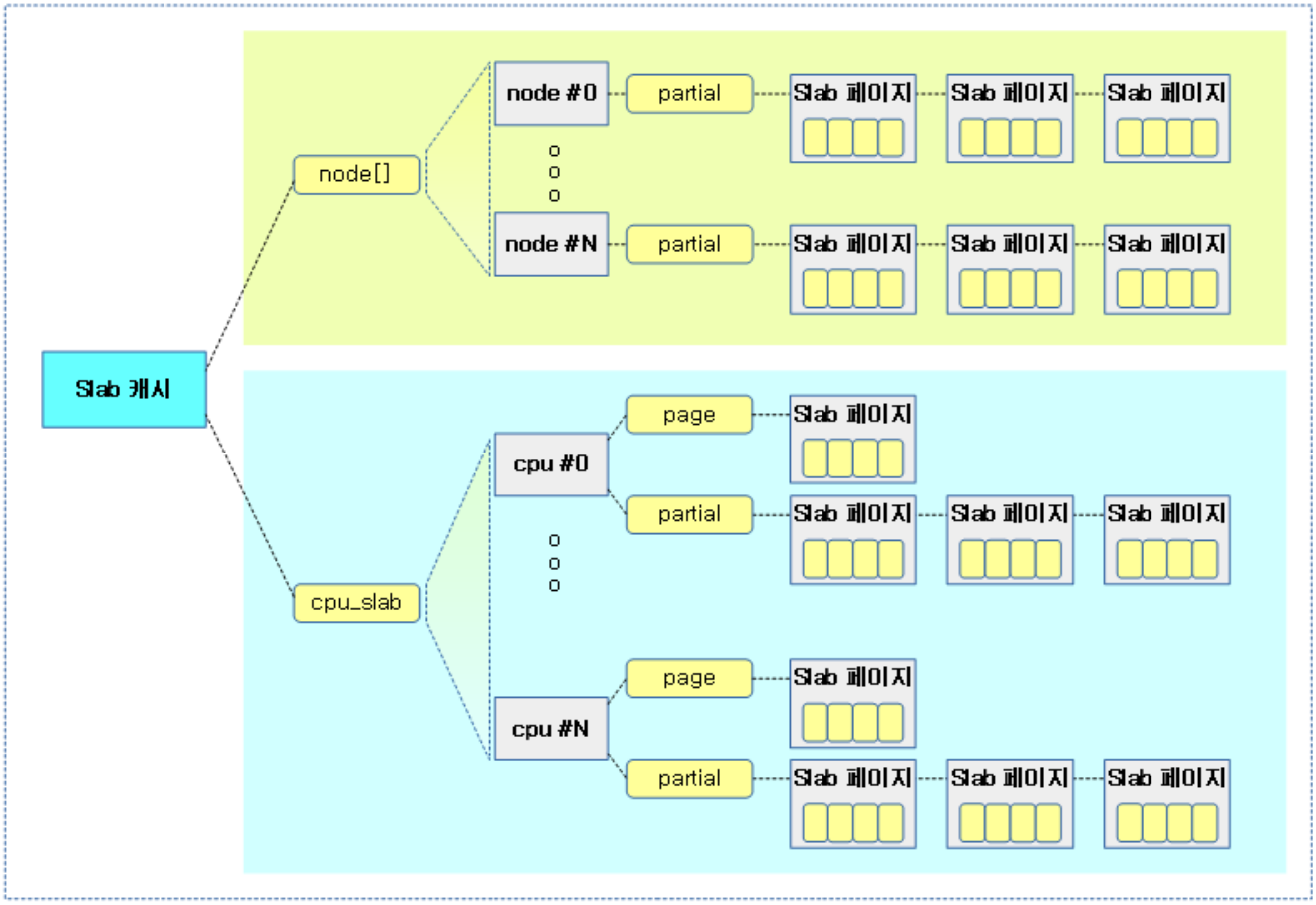
(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab-3.png)

Support for per-node and per-CPU management

- per-node
  - Since the memory access speed is different for each node, the slab pages are managed separately by node.
- per-cpu
  - It is managed separately by CPU for fast slap cache allocation using lock-less. The per-cpu slab cache is specified with a partial list and one page.

The following figure shows how slab pages are managed by node and by CPU.

- Each CPU has a slab page related to allocation/retrieval and the rest is managed in a partial list.

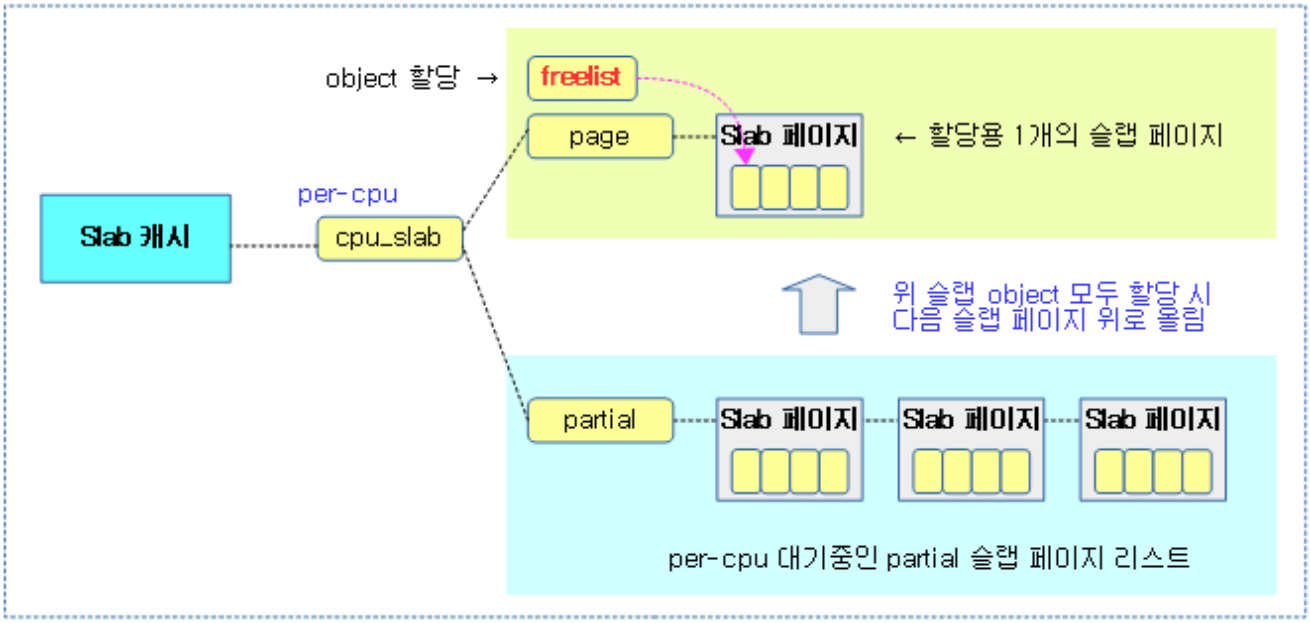


(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab-5.png)

Freelist

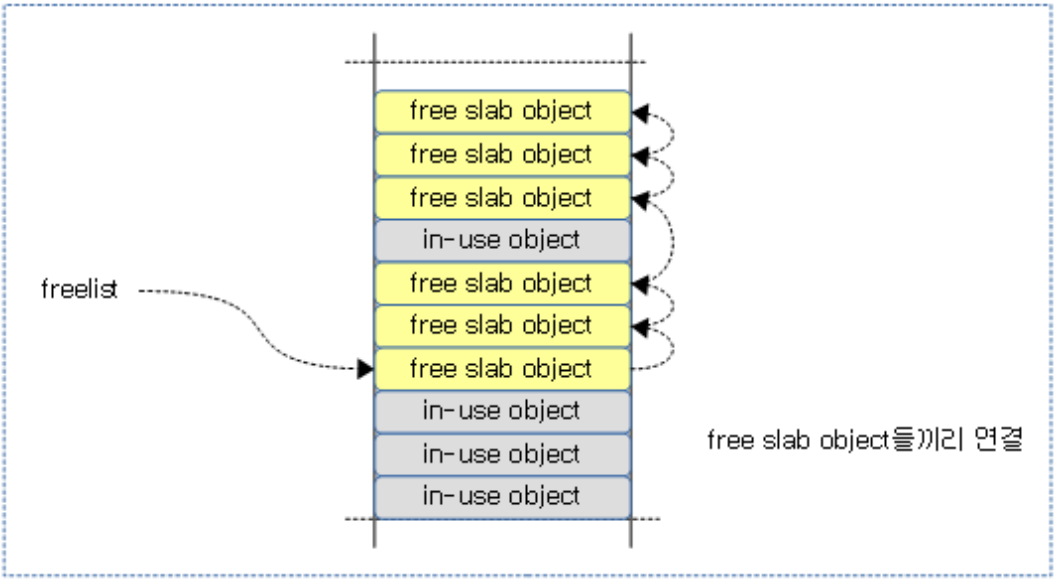
On the per-cpu slab page, it points to the beginning of the free object. Each free object is used as a list of free objects, as it points to the next free object. This freelist can be assigned exclusively to that CPU, and can be turned off for any other CPU.

The following figure shows a slab page for each CPU, pointing to a free object on the slab page specified through freelist.



(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slab-7a.png)

The following illustration shows how the freelist points to the first free slab object, and each free object is connected to each other in order.



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

Items inside a slab object

The items that make up the slab object are as follows.

Object Area

- The minimum size of a slab object is 32 bytes on a 4-bit system and 64 bytes on a 8-bit system.

FP(Free Pointer)

- When each object in the page frame is free, the FP (Free Pointer) at the top of the object points to the next free object. When debugging an object, even if you don't use the object, all the data

in the object is set to uninitialized poison data and is monitored to make sure it hasn't been breached. Therefore, in this case, FP(Free Pointer) is moved behind the object. This is followed by an owner track field and a red zone field that are used for debugging.

- Free objects use FP (Free Pointer) at the top of the object with an offset of 0 to point to the address of the next free object. However, if you used the SLAB_DESTROY_BY_RCU and SLAB_POISON flag options, or if you used a constructor, assign a object_size to offset to move the FP (Free Pointer) back object_size.
- For improved security, you can use CONFIG_SLAB_FREELIST_HARDENED kernel options to encaptualate and hide free pointer values.

Poison

- It is used to detect data address infringement.
- After the destruction of the slab object, the poison value is recorded in the space corresponding to the object_size, and when the slab object is created, the value is detected and reported as an error output.
- The poison value is as follows:
  - If the slab object is in use but not initialized, it is populated with a value of 0x5a='Z'.
  - When the slab object is not used, it fills it with the value 0x6b='k' and only the last byte is 0xa5.
- You can use the "slub_debug=P" kernel parameter to debug poison.
- Debugging notes:
  - slub_debug: Detect Kernel heap memory corruption (http://techvolve.blogspot.com/2014/04/slubdebug-detect-kernel-heap-memory.html) | TechVolve
  - Short users guide for SLUB (https://www.kernel.org/doc/Documentation/vm/slub.txt) (2015, Documentation/vm/slub.txt)| Kernel.org

Red-Zone

- It is used to detect data address infringement.
- After recording the red-zone value on the left and right sides of the object_size, it detects that this value changes when the slab object is created and destroyed, and reports it as an error output.
- The red zone values are as follows:
  - When it is inactive, it is filled with 0xbb values.
  - When active, it is filled with 0xcc values.
- You can use the "slub_debug=Z" kernel parameter for red-zone debugging.

Owner(User) Track

- It is a function that outputs up to 16 functions each called when creating and destroying slab objects.
- You can use the "slub_debug=U" kernel parameter to track users.

Padding

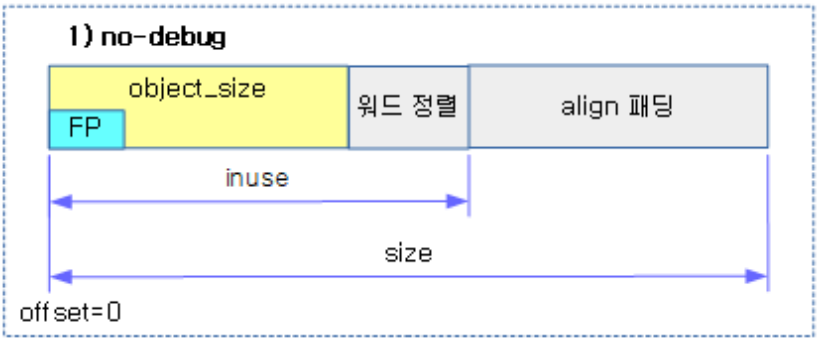
- Align Fill in the 0x5a='Z' with the padding value according to the alignment.

Slap Object Internal Structure

Note: In the figure below, the position of the FP has been moved to the center using the pointer size alignment unit of the object_size.

- Consultation:
  - slub: relocate freelist pointer to middle of object (https://github.com/torvalds/linux/commit/3202fa62fb43087387c65bfa9c100feffac74aa6) (2020, v5.7-rc1)
  - mm/slub: fix redzoning for small allocations (https://github.com/torvalds/linux/commit/74c1d3e081533825f2611e46edea1fcdc0701985) (2021, v5.13-rc7)
  - mm/slub: actually fix freelist pointer vs redzoning (https://github.com/torvalds/linux/commit/e41a49fadbc80b60b48d3c095d9e2ee7ef7c9a8e) (2021, v5.13-rc7)

1) Slub object without meta information

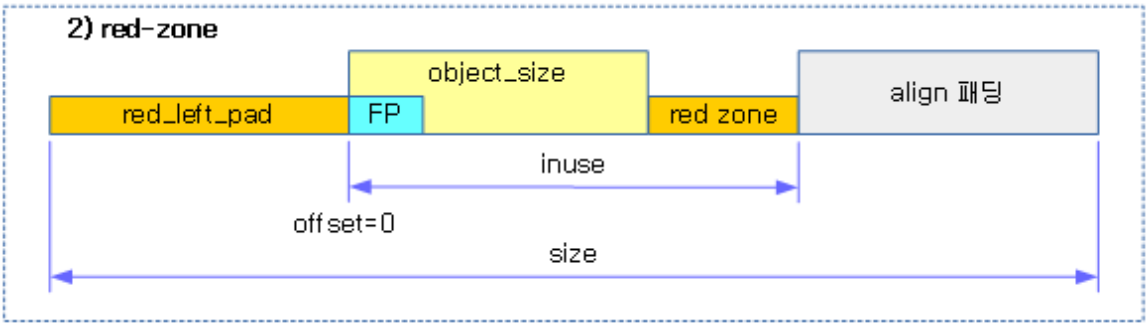


(http://jake.dothome.co.kr/wp-content/uploads/2016/06/slub_object-1g.png)

- The overall size includes the actual object size + alignment of the padding.
  - e.g. object_size=22, align=8
    - inuse=24, size=24
  - e.g. object_size=22, align=64
    - inuse=24, size=64
- The offset for FP is 0.
- The minimum sort size is in words (32bit=4, 64bit=8).
  - e.g. object_size=22, align=0
    - size=24
- SLAB_HWCACHE_ALIGN When using the GPF flag, if the L1 cache line size is smaller than the cash line size, align units are reduced to the order of 2 to minimize cache line bouncing.
  - Examples: ..., 64, 32, 16, 8
- e.g. object_size=22, align=22, flags=SLAB_SWCACHE_ALIGN

- size=32

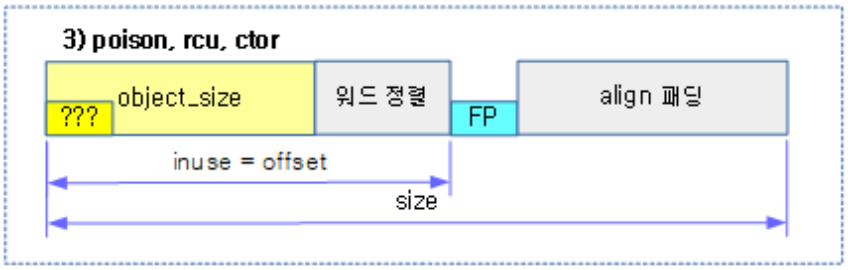
2) Slub Object with Red-Zone Information



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

- Even if the previous object overwrites, we add red_left_pad space to protect the next object.
- Adjacent to the right side of the object is the red zone with a minimum size of 1~maximum word size (32bit=4, 64bit=8).
  - If the object size is already sorted in words and there is no redzone space, add the word size length as a redzone place.
  - In addition, the yellow space, i.e., the starting position of the object for the user, is aligned, and the entire object size is aligned.

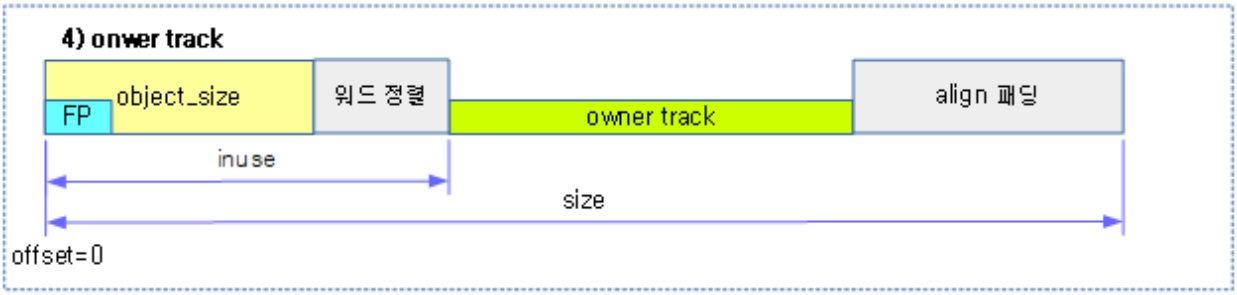
3) Slub Object with FP Movement (Poison, RCU, CTOR)



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

- Poison debugging, which requires separate information to be written to the object location, free object support using RCU, or slab caches using constructors should move the FP (Free Pointer) location to the object next. The offset for FP (Free Pointer) is the same as inuse.
- Here are 3 items that you should move your FP to:
  - If you use the SLAB_POISON flag, you should log the poison data in the following two cases:
    - object is free
    - object is allocated but not initialized
  - If the SLAB_TYPESAFE_BY_RCU flag is used, the pointer to the free function using the RCU is stored.
  - For slab caches where constructors are used, record the constructor function pointer.

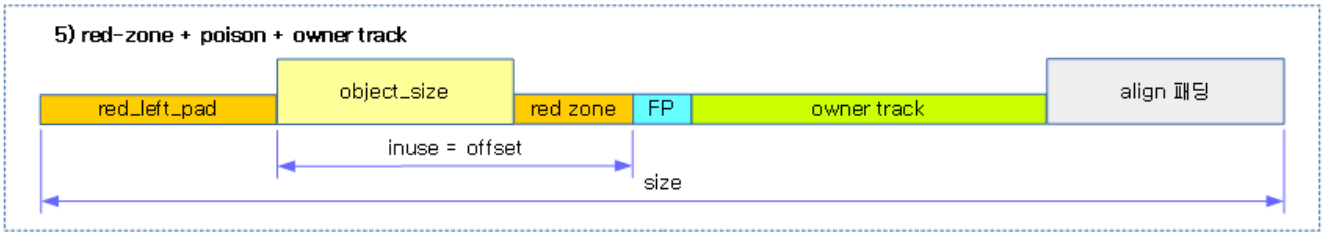
4) SLUB Object containing Owner Track information



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

- Added owner track information to track users who allocate/unassign objects.
  - When using the SLAB_STORE_USER flag, use two track structs for owner track information.

5) Slub Object with Red-Zone + FP Movement + Owner-Track Information



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

- Although not shown in the figure, when debugging KASAN, KASAN-related information is added after the owner track.

Structure

kmem_cache Structure (SLUB)

include/linux/slub_def.h

```
1  /*
2   * Slab cache management.
3   */
```

```
01 struct kmem_cache {
02         struct kmem_cache_cpu __percpu *cpu_slab;
03         /* Used for retriving partial slabs etc */
04         slab_flags_t flags;
05         unsigned long min_partial;
06         unsigned int size;      /* The size of an object including meta
   data */
07         unsigned int object_size;/* The size of an object without meta d
   ata */
08         unsigned int offset;    /* Free pointer offset. */
09 #ifdef CONFIG_SLUB_CPU_PARTIAL
10         /* Number of per cpu partial objects to keep around */
11         unsigned int cpu_partial;
12 #endif
13         struct kmem_cache_order_objects oo;
14
15         /* Allocation and freeing of slabs */
16         struct kmem_cache_order_objects max;
17         struct kmem_cache_order_objects min;
18         gfp_t allocflags;       /* gfp flags to use on each alloc */
```

```
        int refcount;           /* Refcount for slab cache destroy */
        void (*ctor)(void *);
        unsigned int inuse;             /* Offset to metadata */
        unsigned int align;             /* Alignment */
        unsigned int red_left_pad;      /* Left redzone padding size */
        const char *name;       /* Name (only for display!) */
        struct list_head list;  /* List of slab caches */
#ifdef CONFIG_SYSFS
        struct kobject kobj;    /* For sysfs */
        struct work_struct kobj_remove_work;
#endif
#ifdef CONFIG_MEMCG
        struct memcg_cache_params memcg_params;
        /* for propagation, maximum size of a stored attr */
        unsigned int max_attr_size;
#ifdef CONFIG_SYSFS
        struct kset *memcg_kset;
#endif
#endif

#ifdef CONFIG_SLAB_FREELIST_HARDENED
        unsigned long random;
#endif

#ifdef CONFIG_NUMA
        /*
         * Defragmentation by allocating from a remote node.
         */
        unsigned int remote_node_defrag_ratio;
#endif

#ifdef CONFIG_SLAB_FREELIST_RANDOM
        unsigned int *random_seq;
#endif

#ifdef CONFIG_KASAN
        struct kasan_cache kasan_info;
#endif

        unsigned int useroffset;        /* Usercopy region offset */
        unsigned int usersize;          /* Usercopy region size */

        struct kmem_cache_node *node[MAX_NUMNODES];
};
```

Manage the slab cache.

- cpu_slab
  - per-cpu cache
- flags
  - Flag options applied at cache creation
- min_partial
  - Minimum number of partial slap pages to keep in the cache
- size
  - Included object and metadata, aligned size
- object_size
  - The size of the object, excluding metadata.
- offset
  - Offset the position of the Free Pointer (FP) within an object
  - 0 without SLAB_POISON and SLAB_DESTROY_BY_RCU

- cpu_partial
  - Maximum number of slab objects that can be maintained in the per-cpu partial list
- oo
  - Recommended order to apply when creating a slab page
  - If you are not in a low-memory situation, assign the slab page with the recommended order.
  -
- max
  - When creating a slab page, the maximum order
- min
  - When creating a slab page, the minimum order
  - In case of low memory, the slab page is allocated with min order.
- allocflags
  - GFP flags to use when assigning objects
- refcount
  - It is a reference counter to use to delete the slab cache, which continues to grow when the alias cache is created.
- (*ctor)
  - object constructor
- inuse
  - Actual size, excluding space added by metadata.
  - It is the same size as if metadata (SLAB_POISON, SLAB_DESTROY_BY_RCU, SLAB_STORE_USER, KASAN) is not used.
- align
  - Number of bytes to sort
- red_left_pad
  - Left red-zone padding size
- reserved
  - The number of bytes that need to be reserved at the end of the slub object.
- *name
  - Name used for output only
- list
  - A list node used to connect slab caches.
- kobj
  - Directory information to use when creating in sysfs
- kobj_remove_work
  - When deleting the slab cache, it removes the directory of the slab cache name created in sysfs that is linked through work.
- memcg_param
  - Parameters Used by Memory cGroups
- max_attr_size
  - The maximum size at which the attribute will be stored.
- *memcg_kset
  - kset used by memory cgroups

- random
  - CONFIG_SLAB_FREELIST_HARDENED a random value to encaptualize the FP (Free Pointer) for security purposes when using kernel options to hide it from recognition.
- remote_node_defrag_ratio
  - If there are not enough slub objects to be assigned from the local node's partial list, the probability of trying them from the remote node's partail list
  - 0~1023 (100=Tried by remote node about 10% chance when slub object cannot be allocated from the partial list of local nodes)
- *random_seq
  - When using CONFIG_SLAB_FREELIST_RANDOM kernel options, an array is allocated to shuffle the order of free objects for the purpose of increasing security against HEAP overflow intrusions.
- kasan_info
  - CONFIG_KASAN kernel options allow you to use the Kernel Address SANitizer (KASAN) runtime debugger.
- useroffset
  - User copy area offset
- usersize
  - User copy area size
- *node
  - kmem_cache_node array pointer to manage a partial list by node

kmem_cache_cpu Structure (SLUB)

include/linux/slub_def.h

```
01 struct kmem_cache_cpu {
02         void **freelist;        /* Pointer to next available object */
03         unsigned long tid;      /* Globally unique transaction id */
04         struct page *page;      /* The slab from which we are allocating
   */
05 #ifdef CONFIG_SLUB_CPU_PARTIAL
06         struct page *partial;   /* Partially allocated frozen slabs */
07 #endif
08 #ifdef CONFIG_SLUB_STATS
09         unsigned stat[NR_SLUB_STAT_ITEMS];
10 #endif
11 };
```

Per-CPU-managed slap cache

- **freelist
  - A pointer to an assignable free object from any of the page members below.
- tid
  - Globally unique transaction IDs
- *page
  - Slap cache page being used for allocation/release
- *partial
  - List of frozen slab pages where some objects are in-use
- stat

- Slap Cache Stats

kmem_cache_node structs (slab, slub)

mm/slab.h

```
#ifndef CONFIG_SLOB
/*
 * The slab lists for all objects.
 */
```

```
struct kmem_cache_node {
        spinlock_t list_lock;
#ifdef CONFIG_SLUB
        unsigned long nr_partial;
        struct list_head partial;
#ifdef CONFIG_SLUB_DEBUG
        atomic_long_t nr_slabs;
        atomic_long_t total_objects;
        struct list_head full;
#endif
#endif
};
```

Per-node-managed slab cache

- list_lock
  - Used by Spin Lock
- nr_partial
  - Number of partial lists to keep
- partial
  - A list of slab pages partialized on that node
  - However, unlike slab, slub doesn't specifically distinguish between partial states.
- nr_slabs
  - Number of slap pages for debug
- total_objects
  - The total number of slab objects for debugging
- full
  - List of in-use slub pages for debugging

A member used for slub purposes in a page structure.

include/linux/mm_types.h

```
struct page {
...
                struct {        /* slab, slob and slub */
                        union {
                                struct list_head slab_list;     /* uses
lru */
                                struct {        /* Partial pages */
                                        struct page *next;
#ifdef CONFIG_64BIT
                                        int pages;      /* Nr of pages l
eft */
```

```
10                                      int pobjects;   /* Approximate c
ount */
11  #else
12                                      short int pages;
13                                      short int pobjects;
14  #endif
15                              };
16                      };
17                      struct kmem_cache *slab_cache; /* not slob */
18                      /* Double-word boundary */
19                      void *freelist;         /* first free object */
20                      union {
21                              void *s_mem;    /* slab: first object */
22                              unsigned long counters;         /* SLUB
*/
23                              struct {                        /* SLUB
*/
24                                      unsigned inuse:16;
25                                      unsigned objects:15;
26                                      unsigned frozen:1;
27                              };
28                      };
29              };
30  ...
```

This is a page descriptor used for slab pages.

- slab_list
  - Nodes to be used when connecting to the per-cpu cache or to a partial list of per-nodes
- *next
  - Point to the following partial page:
- pages
  - Number of partial pages remaining
- pobjects
  - Approximate number of objects remaining
- *slab_cache
  - Slap Cache
- *freelist
  - A pointer pointing to the first free object.
- counters
  - It contains the following information:
    - inuse:16
      - The total number of objects in use
      - counters[15:0]
    - objects:15
      - Total number of objects
      - counters[30:16]
    - frozen:1
      - Indicates whether it is frozen or not.
      - counters[31]

Frozen status of the slab page

- The state in which the slab page is available exclusively to a particular CPU is frozen.
  - Slab pages linked to c->page, or slab pages linked to c->partial, are in the frozen state.
- The slab pages in the partial list management by node are un-frozen.
  - The slab pages linked to node[]->partial are in an un-frozen state.
- A dedicated CPU can browse and allocate/decommission slab objects from the freelist of frozen pages.
- CPUs other than the dedicated CPU are not allowed to browse and allocate slab objects from the freelist, and are only allowed to deallocate slab objects.

kmem_cache_order_objects Struct

include/linux/slub_def.h

```
/*
 * Word size structure that can be atomically updated or read and that
 * contains both the order and the number of objects that a slab of the
 * given order would contain.
 */
```

```
struct kmem_cache_order_objects {
        unsigned long x;
};
```

- x[15:0]
  - Maximum number of objects to use when creating a slub page
- x[31:16]
  - order to use when creating a slub page

consultation

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

- Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB (Rev. 2014) | Christoph Lameter – pdf download (https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf)
- A Heap of Trouble (Exploiting the Linux Kernel SLOB Allocator) | Dan Rosenberg – pdf download (http://vulnfactory.org/research/slob.pdf)
- How does the SLUB allocator work | Junsu Kim – PDF Download (https://events.static.linuxfound.org/images/stories/pdf/klf2012_kim.pdf)
- [Linux] dynamic memory allocator: slab, slub, slob (http://egloos.zum.com/studyfoss/v/5332580) | F/OSS
- The SLUB Allocator (http://lwn.net/Articles/229984/) | LWN.net
- Kernel dynamic memory analysis (http://elinux.org/Kernel_dynamic_memory_analysis) | elinux.org
- SLUB: Support for statistics to help analyze allocator behavior (https://lwn.net/Articles/267834/) | LWN.net
- Cramming more into struct page (https://lwn.net/Articles/565097/) | LWN.net
- debug with Linux slub allocator (http://thinkiii.blogspot.kr/2014/02/debug-with-slub-allocator.html) | thinkiii
- slub_debug: Detect Kernel heap memory corruption (http://techvolve.blogspot.com/2014/04/slubdebug-detect-kernel-heap-memory.html) | TechVolve
- Short users guide for SLUB (https://www.kernel.org/doc/Documentation/vm/slub.txt) (2015, Documentation/vm/slub.txt)| Kernel.org

---

8 thoughts to "Slab Memory Allocator -1- (Structure)"

JAEHO JUNG
2021-02-03 01:47 (http://jake.dothome.co.kr/slub/#comment-304503)

Hello and thank you for the good article. I have a question and leave it.
I don't understand the part that says "the per-cpu slab cache is specified with a partial list and 1 page". It was understood that each slap page linked to a partial list contained objects that were internally either free or inuse. So, what does this part mean when "1 page is specified"?
In the article, there is an explanation that 'slab pages related to allocation/recall are specified for each CPU', but I don't understand that part. What is the difference between a slab page linking to a partial list and a single slab page linking to a page?

Just looking at the picture, they are both connected by the same type of slab page.

RESPONSE (/SLUB/?REPLYTOCOM=304503#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-02-03 06:50 (http://jake.dothome.co.kr/slub/#comment-304506)

Hello?

In the partial list, the slab pages are in the c->partial list, and they are waiting. You can't allocate a slab object directly from here.
The slab page to which the actual slab object can be assigned is taken from one of the c->partial lists above and assigned to c->page.

In other words, the place where the srep pages wait in the CPU unit is the c->partial list, and only one of the slab pages is specified in the c->page.

The freelist illustration in the text has been modified to make it easier to understand.

I appreciate it.

RESPONSE (/SLUB/?REPLYTOCOM=304506#RESPOND)

JAEHO JUNG
2021-02-03 21:35 (http://jake.dothome.co.kr/slub/#comment-304511)

1. Thank you!. To sum it up, in order for the slab pages in the c->partial list (which have objects to be free or inuse) to be used, a single slab page must be uploaded to c->page and the free objects contained in it must be managed through the freelist! That's right, right?

2. In addition, I have a question about managing slab pages in kmem_cache structure with per-cpu and per-node. I understood that per-cpu literally manages the slap cache on a per-CPU basis to increase speed, just like in number 1 above. So how do we understand per-node? (The concept of node itself is a bundle of CPU and local memory as a unit called a node.)
If the CPU is included in the node unit, why is there a reason to manage per-node and per-cpu separately? I don't know what the difference is between the two.

I appreciate it

RESPONSE (/SLUB/?REPLYTOCOM=304511#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-02-04 16:34 (http://jake.dothome.co.kr/slub/#comment-304532)

You've seen exactly number 1.
Answer question 2.
Per-CPU manages only a few pages, and the rest of the slab pages are mostly managed by per-node.

In addition to the slab pages prepared by per-cpu, we need a list where more slab pages are pre-formatted,
which is divided into per-node lists.

Of course, the reason why we separated it into per-nodes is to request the slab pages that per-cpu needs to be a little faster.

For example, in a system of 128 NUMA nodes with 2 CPUs, each CPU has a few slap pages,
and everything else is ready in the per-node.

슬랩 object의 할당은 다음 순서대로 이루어집니다. 부족할 때마다 좌측으로 refill 한다고 생각하시면 됩니다.
슬랩 object <- per-cpu 페이지의 free list <- per-cpu partial 리스트 <- per-node 리스트 <- kmalloc(버디) 감사합니다.

응답 (/SLUB/?REPLYTOCOM=304532#RESPOND)

정재호
2021-02-04 22:10 (http://jake.dothome.co.kr/slub/#comment-304534)

감사합니다 헷갈리던 부분 확실하게 이해하고 넘어갑니다ㅎㅎㅎ

응답 (/SLUB/?REPLYTOCOM=304534#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2021-02-05 13:53 (http://jake.dothome.co.kr/slub/#comment-304536)

좋은 하루되세요. ^^

응답 (/SLUB/?REPLYTOCOM=304536#RESPOND)

이대로
2021-10-25 10:07 (http://jake.dothome.co.kr/slub/#comment-306086)

안녕하세요
Order N 페이지에 대해 궁금한 점이 있어 질문남깁니다.

‘슬랩 페이지 내 object 배치’ 그림에서 ‘Order N 페이지’에서 ‘Object’ 크기만큼 나누어서 Object를 만드는 데, ‘Order N 페이지’ 내의 각각의 페이지가 struct page 구조체의 slub 용도로 사용되는 멤버로 관리가 되는 것인가요?(예를 들면, N=3 일 때 8개의 페이지가 각각 struct page 구조체로 관리) 아니면 ‘Order N 페이지’를 하나의 struct page 구조체가 관리하는 것인까요?(N=3일 때, 8개의 페이지 전체가 하나의 struct page 구조체로 관리)

응답 (/SLUB/?REPLYTOCOM=306086#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2021-10-25 13:33 (http://jake.dothome.co.kr/slub/#comment-306087)

안녕하세요?

버디 시스템에서 슬랩 캐시에 슬랩 페이지들을 할당할 때, 가급적 custom 드라이버등을 제외하곤 코어 레벨에서 PAGE_ALLOC_COSTLY_ORDER(3) ~~마만~~이하의 order로 페이지들을 할당하여 사용하는 것으로 단편화를 회피하고 있습니다. 이 때 슬랩 캐시에서 사용하는 슬랩 페이지가 order 0를 제외한 order N으로 할당하는 경우 compound 페이지 형식을 따릅니다. 다음 URL을 참고하시길 바랍니다.
http://jake.dothome.co.kr/compound/ (http://jake.dothome.co.kr/compound/)

감사합니다.

응답 (/SLUB/?REPLYTOCOM=306087#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

❮ Slub Memory Allocator -4- (order 계산) (http://jake.dothome.co.kr/slub-order/)

Slub Memory Allocator -3- (캐시 생성) ❯ (http://jake.dothome.co.kr/slub-cache-create/)

문c 블로그 (2015 ~ 2023)