Vmap

2016-07-22 (http://jake.dothome.co.kr/vmap/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Vmap

The kernel is very reluctant to use high-order page allocation, which causes fragmentation problems. Therefore, if you need to allocate large pages but do not allocate/undo them frequently, you can use multiple single (order 0) pages and map them together in a contiguous virtual address space. This mapping method is called vmap (Virtually contiguous memory area mapping).

Here's an API that uses vMap to allocate memory to two address spaces:

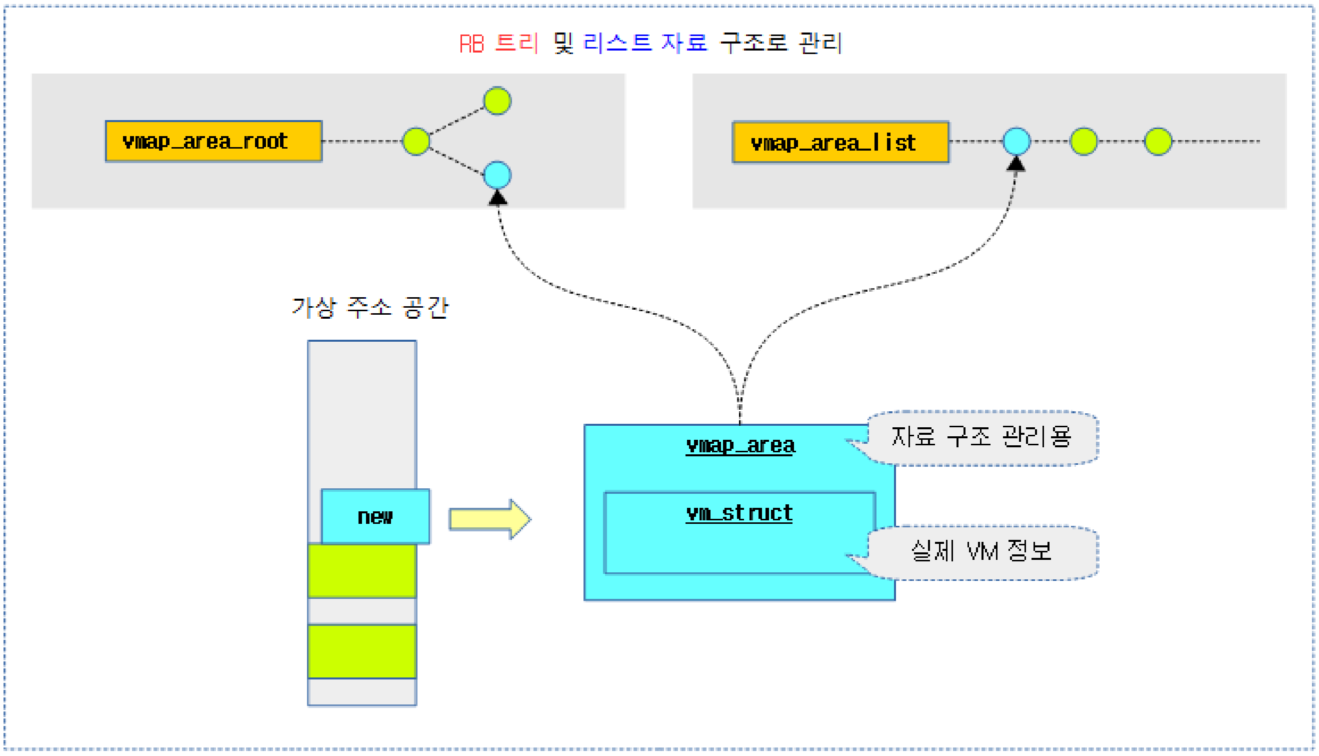
- vmalloc()
  - Assign as many pages as the request size to map to the vmalloc address space space
- module_alloc()
  - Assign as many pages as the request size and map them to the module address space

Manage VM Zones

For searching for empty spaces in the VMALLOC virtual address space, the RB tree and list data structure are used simultaneously.

- RB Tree
- List

The following figure shows VM zones managed in the RB tree and list within the vmalloc virtual address space.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vmap-3.png)

Management Items

Using the above data structure, the items for managing the space in vmalloc are as follows:

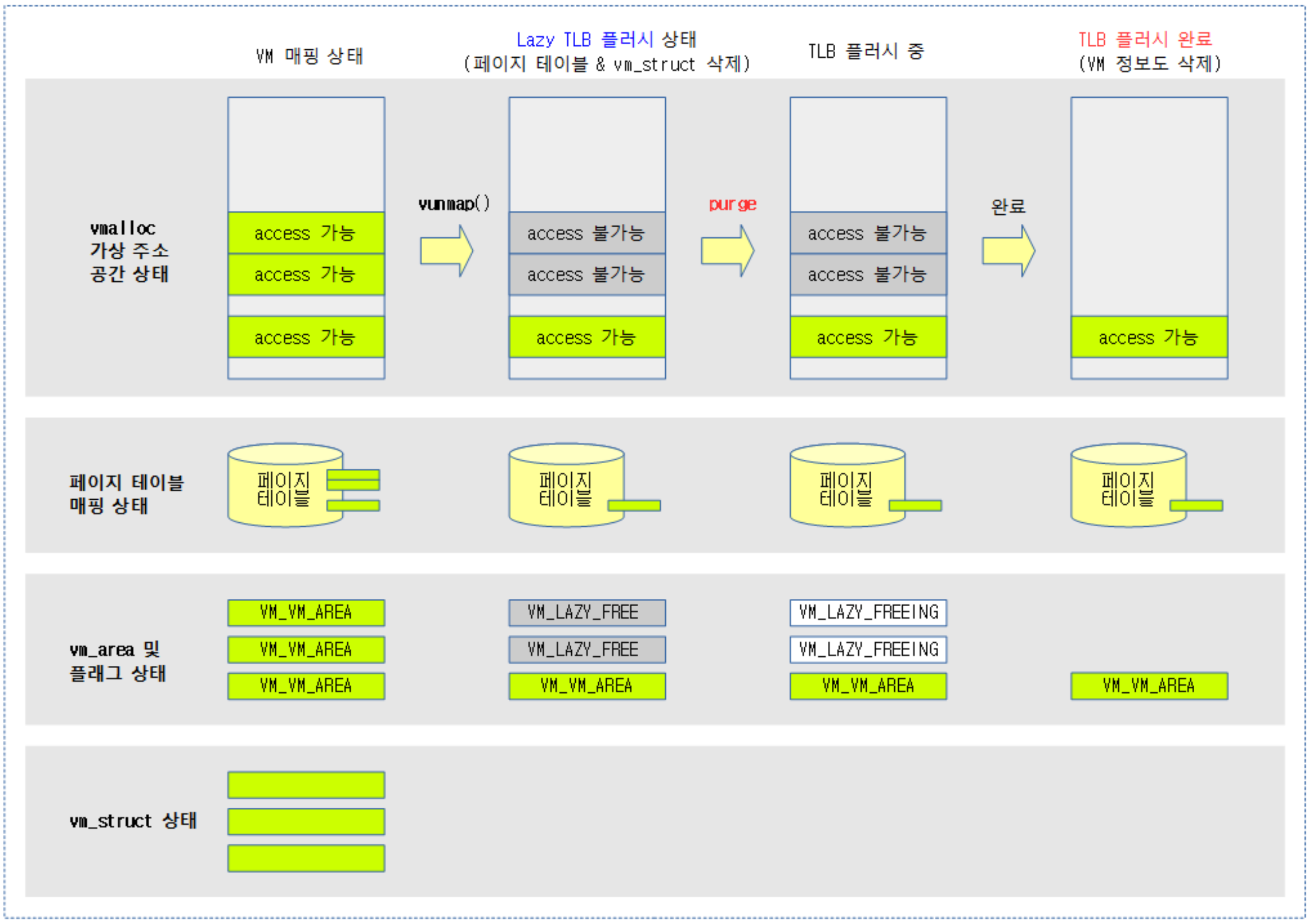
- Manage your footprint
  - vmap_area_root
  - vmap_area_list
- Empty space management
  - ree_vmap_area_root
  - free_vmap_area_list
  - This item was added in kernel v5.2-rc1.
    - 참고: mm/vmalloc.c: keep track of free blocks for vmap allocation (https://github.com/torvalds/linux/commit/68ad4a3304335358f95a417f2a2b0c909e5119c4) (2019, v5.2-rc1)
- Lazy Free Management
  - purge_vmap_area_root
  - purge_vmap_area_list

Lazy TLB Flushing(Free)

- When you perform vunmap(), the following items should be processed: sort out which items are processed immediately and which items are collected later.
  - Immediate Processing
    - Unmap from a page table
    - cache flush
  - Delay Handling

        - Removal of delayed vmap_area from RB trees and lists
        - TLB flush on all CPUs
            - On systems with a lot of CPUs, arm64 has slightly better requirements than arm32 because it can perform TLB flush on CPUs in the inner region in batches. In comparison, in the case of the arm32 system, an IPI (Inter Process Interrupt) call is used to generate an interrupt to each CPU before processing it, which further reduces processing performance.
- It leaves the VMAs to be deleted unprocessed, and when a certain amount is exceeded or the memory is insufficient, the purge is processed all at once to increase processing performance.
    - By using this method of withholding revocation, an implementation was applied that processes the management of vmap_area about 20 times faster.
- The flag bits that represent the lazy TLB free status are as follows:
    - VM_LAZY_FREE (0x1)
        - Deletion requested status
    - VM_LAZY_FREEING (0x2)
        - purge in progress
    - VM_VM_AREA (0x4)
        - Assignment status

The following figure shows how a VM region is changed to a Lazy TLB flush state by vunmap(), and then the actual TLB flushed when purged.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vmap-4b.png)

Related APIs

- vmap()
- vunmap()

Alternate APIs

As an alternative to vmap(), per-cpu map-based APIs have been introduced. This API is not yet used by many drivers, but only by a few drivers.

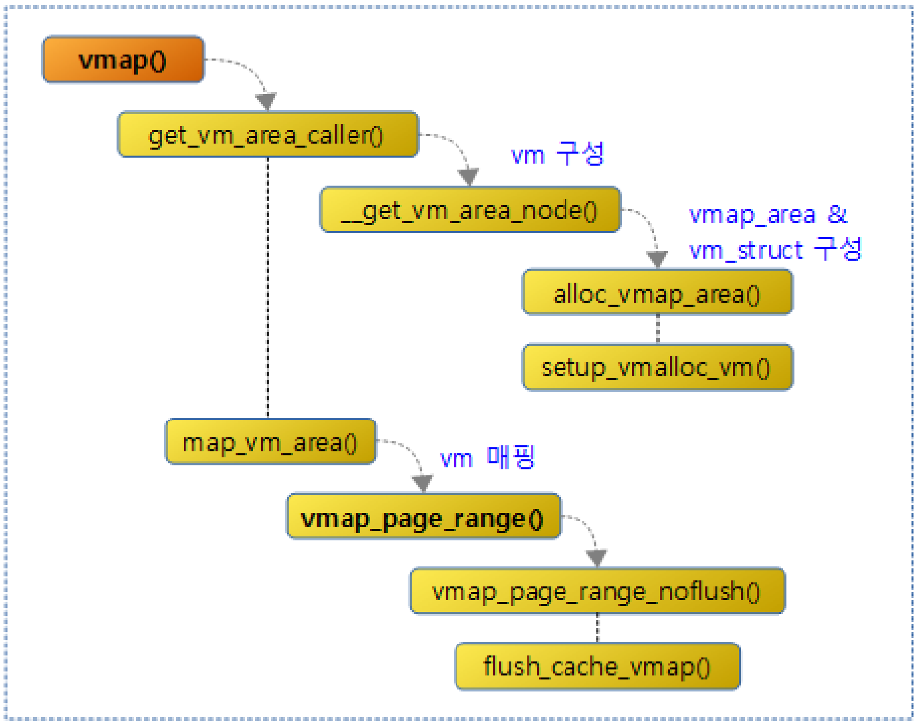
- vm_map_ram()
- vm_unmap_ram()

VMAP mapping

To use the vmap() function, we request a single (order 0) page of physical memory information to be used for mapping, consisting of an array of page descriptors. It then finds an empty spot in the vmalloc virtual address space, maps the pages to the given mapping properties, and returns the mapped virtual start address.

- Use cache nodes and a few variables for faster allocation/release.
  - free_vmap_cache
    - To speed up the search for vacant spaces, the most recently registered and used vmap_area or the most recent free vmap_area of prev vmap_area are retained.
  - cached_hole_size
    - Remember the size of the hole directly in front of the cache.
    - If this value is 0, the search is performed from the beginning without using the cache.
  - cached_vstart
    - Cached start virtual address
  - cached_align
    - Cached align value
- Variables used by per-cpu
  - vmap_area_pcpu_hole
    - It starts at the end of the vmalloc space as the start address of the per-cpu's allocation. (Default value is VMALLOC_END)
    - vmalloc() allocates the address of the vmalloc space from the beginning, but per-cpu does the opposite.

The following figure shows the processing flow of the vmap() function with its related functions.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vmap-2a.png)

vmap()

mm/vmalloc.c

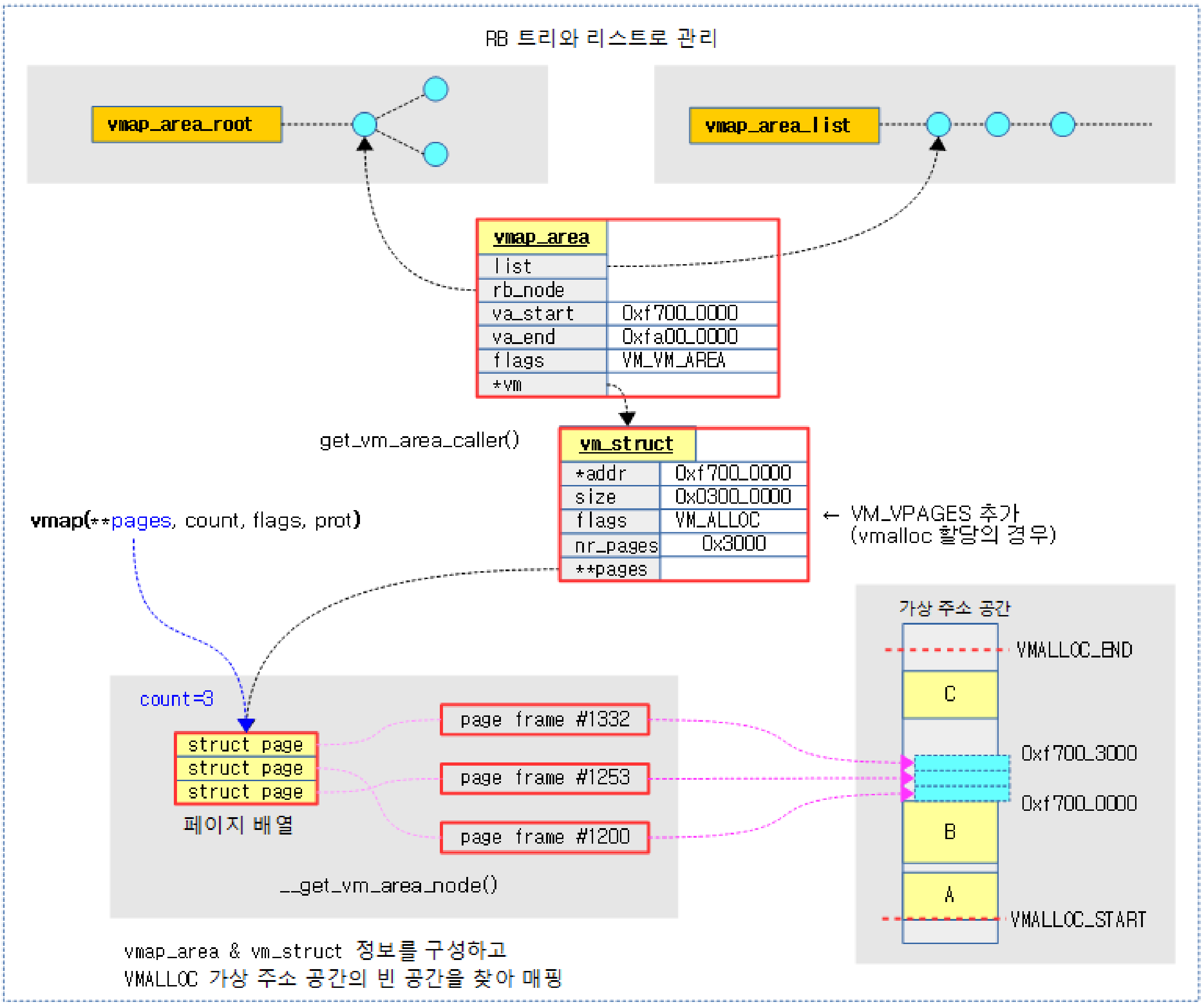
```
/**
 *      vmap  -  map an array of pages into virtually contiguous space
 *      @pages:         array of page pointers
 *      @count:         number of pages to map
 *      @flags:         vm_area->flags
 *      @prot:          page protection for the mapping
 *
 *      Maps @count pages from @pages into contiguous kernel virtual
 *      space.
 */
```

```
void *vmap(struct page **pages, unsigned int count,
                unsigned long flags, pgprot_t prot)
{
        struct vm_struct *area;
        unsigned long size;             /* In bytes */

        might_sleep();

        if (count > totalram_pages())
                return NULL;

        size = (unsigned long)count << PAGE_SHIFT;
        area = get_vm_area_caller(size, flags, __builtin_return_address
(0));
        if (!area)
                return NULL;

        if (map_vm_area(area, prot, pages)) {
                vunmap(area->addr);
                return NULL;
        }

        return area->addr;
}
EXPORT_SYMBOL(vmap);
```

It maps the requested physical pages to a contiguous virtual address space and returns the mapped virtual address. Returns null on failure.

- If you use the CONFIG_PREEMPT_VOLUNTARY kernel option in line 7 of code, sleep if there is a task that you urgently request to be rescheduled with a preempt point.
- If lines 9~10 of the code require more pages than the full memory pages, it will give up processing and return null.
- In code lines 12~15, configure the vmap_area and vm_struct information for VM allocation. If it fails, it returns null.
- In line 17~20 of the code, it tries to map the pages with the vm_struct information, and if it fails, it returns null after freezing.
- Returns the starting address of the virtual address space mapped in line 22 of code.

The following figure shows the mapping of requested physical pages by finding an empty space in the VMALLOC virtual address space.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vmap-1c.png)

Assign a virtual address zone (VM)

get_vm_area_caller()

mm/vmalloc.c

```c
struct vm_struct *get_vm_area_caller(unsigned long size, unsigned long flags,
                                const void *caller)
{
        return __get_vm_area_node(size, 1, flags, VMALLOC_START, VMALLOC_END,
                                  NUMA_NO_NODE, GFP_KERNEL, caller);
}
```

Configure the VM (vm_area and vm_struct) information by finding an empty space in the VMALLOC virtual address space with the requested size (in bytes sorted by pages).

__get_vm_area_node()

mm/vmalloc.c

```c
01 static struct vm_struct *__get_vm_area_node(unsigned long size,
02                 unsigned long align, unsigned long flags, unsigned long start,
03                 unsigned long end, int node, gfp_t gfp_mask, const void *caller)
04 {
05         struct vmap_area *va;
06         struct vm_struct *area;
07 
08         BUG_ON(in_interrupt());
09         size = PAGE_ALIGN(size);
10         if (unlikely(!size))
11                 return NULL;
12 
13         if (flags & VM_IOREMAP)
14                 align = 1ul << clamp_t(int, get_count_order_long(size),
15                                        PAGE_SHIFT, IOREMAP_MAX_ORDER);
16 
17         area = kzalloc_node(sizeof(*area), gfp_mask & GFP_RECLAIM_MASK, node);
18         if (unlikely(!area))
19                 return NULL;
20 
21         if (!(flags & VM_NO_GUARD))
22                 size += PAGE_SIZE;
23 
24         va = alloc_vmap_area(size, align, start, end, node, gfp_mask);
25         if (IS_ERR(va)) {
26                 kfree(area);
27                 return NULL;
28         }
29 
30         setup_vmalloc_vm(area, va, flags, caller);
31 
32         return area;
33 }
```

@start ~ @end find @size of the empty space in the virtual address space of the @align condition, allocate and configure the VM, and return it.

- In line 9~11 of the code, sort the size by page.

- If ioremap is requested in code lines 13~15, use the align value that converts the size to the order unit. However, ioremap is limited to the maximum number of order pages.
  - IOREMAP_MAX_ORDER
    - ARM32 pmd units (16M) bits = > 24
    - If you are using ARM64 4K pages, pud units (1G) bits => 30
    - If you are using ARM64 16K pages, pmd units (32M) bits = > 25
    - If you are using ARM64, 64K pages, pmd units (512M) bits = > 29
- Assign vm_struct in code lines 17~19.
- If there is no guard request in line 21~22 of the code, add 1 page for the guard page.
- Find the empty mapping space in the virtual address range requested in line 24~28 of the code, allocate and configure the vmap_area, insert it into the RB tree and list, and return the entry information.
- On line 30 of code, construct the vm_struct & vm_area struct.
- Returns a pointer to the vm_struct struct configured in line 32 of code.

Assign vmap_area after space discovery

alloc_vmap_area()

mm/vmalloc.c -1/2-

```
1  /*
2   * Allocate a region of KVA of the specified size and alignment, within the
3   * vstart and vend.
4   */
```

```
01 static struct vmap_area *alloc_vmap_area(unsigned long size,
02                                 unsigned long align,
03                                 unsigned long vstart, unsigned long vend,
04                                 int node, gfp_t gfp_mask)
05 {
06         struct vmap_area *va;
07         struct rb_node *n;
08         unsigned long addr;
09         int purged = 0;
10         struct vmap_area *first;
11 
12         BUG_ON(!size);
13         BUG_ON(offset_in_page(size));
14         BUG_ON(!is_power_of_2(align));
15 
16         might_sleep();
17 
18         va = kmalloc_node(sizeof(struct vmap_area),
19                         gfp_mask & GFP_RECLAIM_MASK, node);
20         if (unlikely(!va))
21                 return ERR_PTR(-ENOMEM);
22 
23         /*
24          * Only scan the relevant parts containing pointers to other objects
25          * to avoid false negatives.
26          */
27         kmemleak_scan_area(&va->rb_node, SIZE_MAX, gfp_mask & GFP_RECLAIM_MASK);
28 
```

```
retry:
        spin_lock(&vmap_area_lock);
        /*
         * Invalidate cache if we have more permissive parameters.
         * cached_hole_size notes the largest hole noticed _below_
         * the vmap_area cached in free_vmap_cache: if size fits
         * into that hole, we want to scan from vstart to reuse
         * the hole instead of allocating above free_vmap_cache.
         * Note that __free_vmap_area may update free_vmap_cache
         * without updating cached_hole_size or cached_align.
         */
        if (!free_vmap_cache ||
                        size < cached_hole_size ||
                        vstart < cached_vstart ||
                        align < cached_align) {
nocache:
                cached_hole_size = 0;
                free_vmap_cache = NULL;
        }
        /* record if we encounter less permissive parameters */
        cached_vstart = vstart;
        cached_align = align;

        /* find starting point for our search */
        if (free_vmap_cache) {
                first = rb_entry(free_vmap_cache, struct vmap_area, rb_node);
                addr = ALIGN(first->va_end, align);
                if (addr < vstart)
                        goto nocache;
                if (addr + size < addr)
                        goto overflow;

        } else {
                addr = ALIGN(vstart, align);
                if (addr + size < addr)
                        goto overflow;

                n = vmap_area_root.rb_node;
                first = NULL;

                while (n) {
                        struct vmap_area *tmp;
                        tmp = rb_entry(n, struct vmap_area, rb_node);
                        if (tmp->va_end >= addr) {
                                first = tmp;
                                if (tmp->va_start <= addr)
                                        break;
                                n = n->rb_left;
                        } else
                                n = n->rb_right;
                }

                if (!first)
                        goto found;
        }
```

Find an empty space in the requested virtual address range, allocate and configure va(vmap_area), insert it into the RB tree and list, and return va(vmap_area).

- It first searches the vmap cache to see if it can be reused.
    - If you search from the prev VM of the last registered VM or the last free VM, you can expect quick success.
- If it doesn't find it in the cache, it looks for the entry just above the request start range in the global vmap_area_root configured by the RB tree.

- Next, look for an empty space in the vmap_area_list you made up of lists.
- In the empty space found, configure the memory allocated vmap_area separately and insert it.
- If a space is not found in one search, it flush the lazy vmap_area and search again to find the space.

- In line 18~21 of the code, to construct the vmap_area struct, it receives an allocation using only the reclaim-related flag and returns -ENOMEM if it is an allocation error.
- In code lines 29~47, the retry: label is. If you get a spin-lock and the cached node location is not available, you will not be able to use the cache for this retrieval. The conditions are as follows:
    - If a hole in the cache prev space can cover the newly requested size.
    - The range of the initiated request is less than the scope of the request when using the cache.
    - The requested align value is less than the cached align value.
- In line 53~59 of the code, substitute the rb node that the free_vmap_cache cache points to first, and assign the end address of that node to addr to prepare the search from here.
    - Retrieve VM entries from the free_vmap_cache that holds the last registered or recently free VA (vmap_area) or prev VA.
    - If the end address of the first entry is outside the scope of the request, it is moved to the nocache label to disable the cache.
    - It also adds size to the address at the end of the first entry, and if it exceeds the range, it goes to the overflow label.
- In code lines 61~83, if the free_vmap_cache is not in the cache, it finds the first va in the request scope through the global vmap_area_root RB tree and assigns it to first.
    - If a free_vmap_cache is not available, add size to the address at the end of the first entry, and move to the overflow label if the range is exceeded.

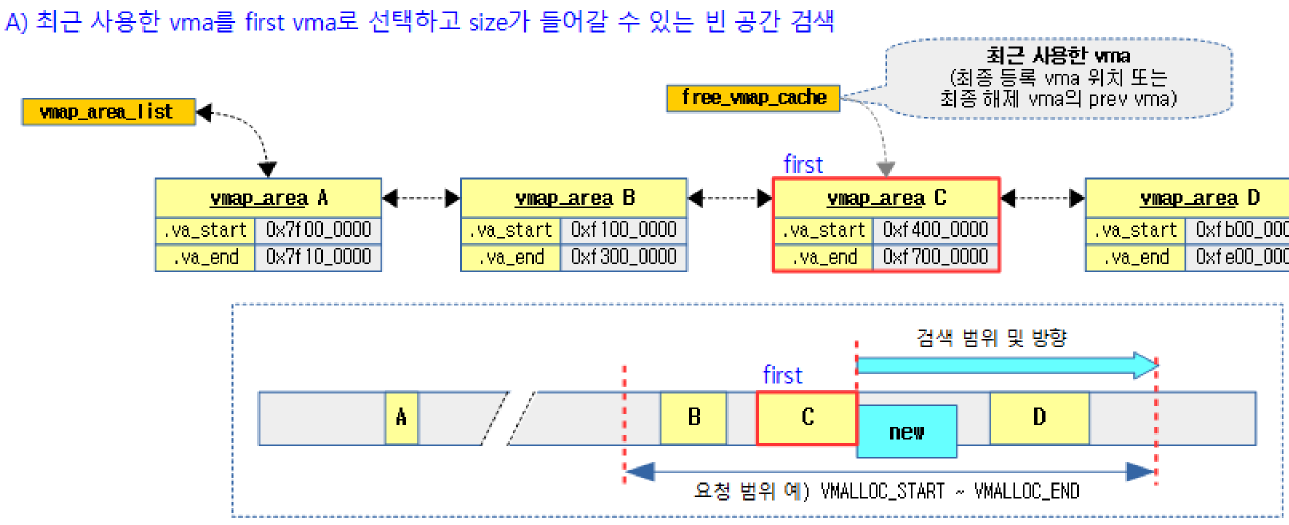
mm/vmalloc.c -2/2-

```c
        /* from the starting point, walk areas until a suitable hole is found */
        while (addr + size > first->va_start && addr + size <= vend) {
                if (addr + cached_hole_size < first->va_start)
                        cached_hole_size = first->va_start - addr;
                addr = ALIGN(first->va_end, align);
                if (addr + size < addr)
                        goto overflow;

                if (list_is_last(&first->list, &vmap_area_list))
                        goto found;

                first = list_next_entry(first, list);
        }

found:
        if (addr + size > vend)
                goto overflow;

        va->va_start = addr;
        va->va_end = addr + size;
        va->flags = 0;
        __insert_vmap_area(va);
```

```
23          free_vmap_cache = &va->rb_node;
24          spin_unlock(&vmap_area_lock);
25
26          BUG_ON(!IS_ALIGNED(va->va_start, align));
27          BUG_ON(va->va_start < vstart);
28          BUG_ON(va->va_end > vend);
29
30          return va;
31
32  overflow:
33          spin_unlock(&vmap_area_lock);
34          if (!purged) {
35                  purge_vmap_area_lazy();
36                  purged = 1;
37                  goto retry;
38          }
39
40          if (gfpflags_allow_blocking(gfp_mask)) {
41                  unsigned long freed = 0;
42                  blocking_notifier_call_chain(&vmap_notify_list, 0, &freed);
43                  if (freed > 0) {
44                          purged = 0;
45                          goto retry;
46                  }
47          }
48
49          if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit())
50                  pr_warn("vmap allocation for size %lu failed: use vmalloc=<size> to increase size\n",
51                          size);
52          kfree(va);
53          return ERR_PTR(-EBUSY);
54  }
```

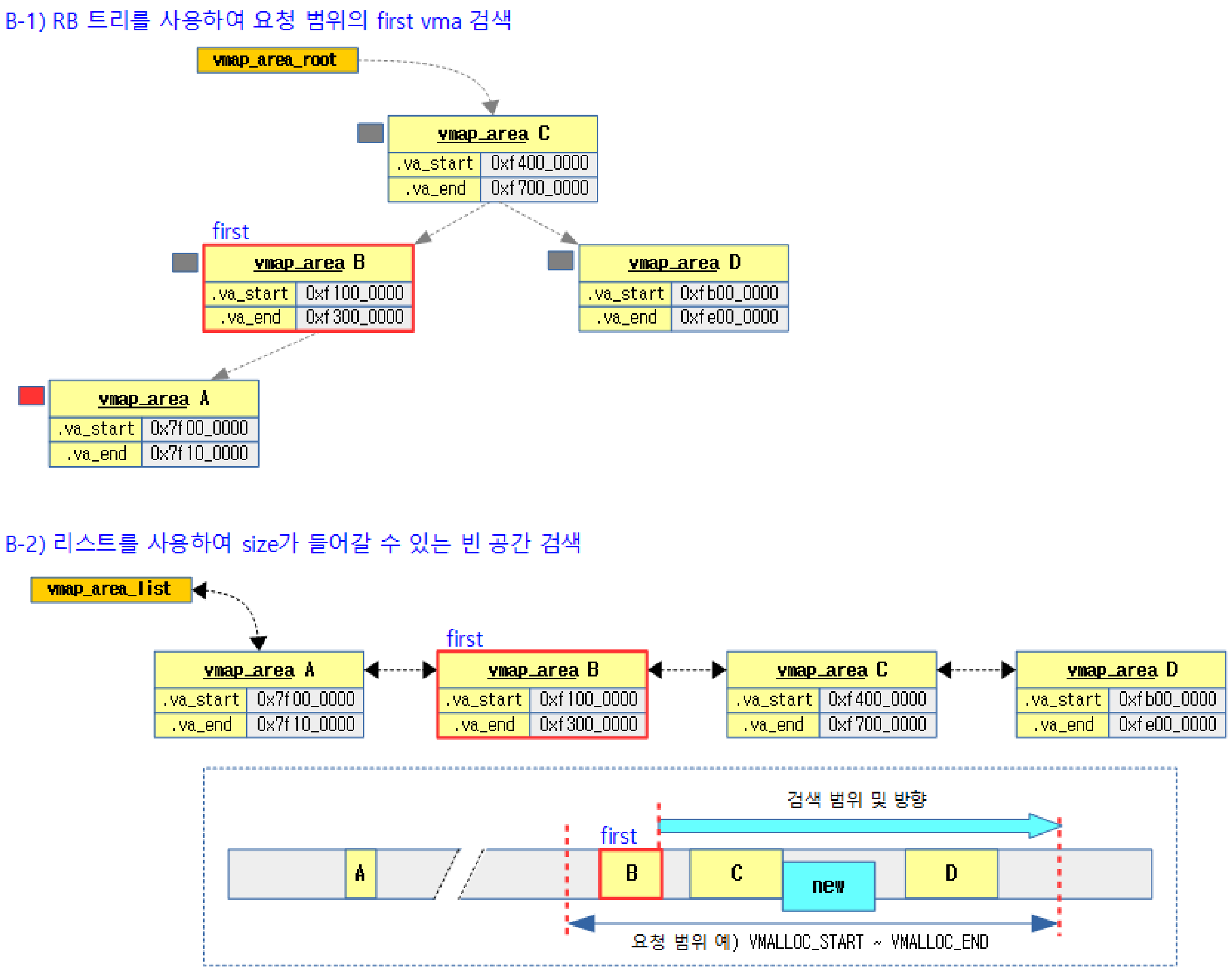
- In code lines 2~13, find an empty space where size can fit within the scope of the request from the global first va (vmap_area) to the end of the list.
- In line 15~30 of the code, it is the found: label. If it finds a suitable space, insert it into the RB tree and list, and return the area.
- In code lines 32~47, the overflow: label is. If no empty space is found, purge all the VM entries in the free state with lazy TLB flush and delete them and try again only once.

The following figure shows how when looking for an empty space within the request virtual address range for mapping, it first searches for empty space for size from free_vmap_cache.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_vmap_area-1b.png)

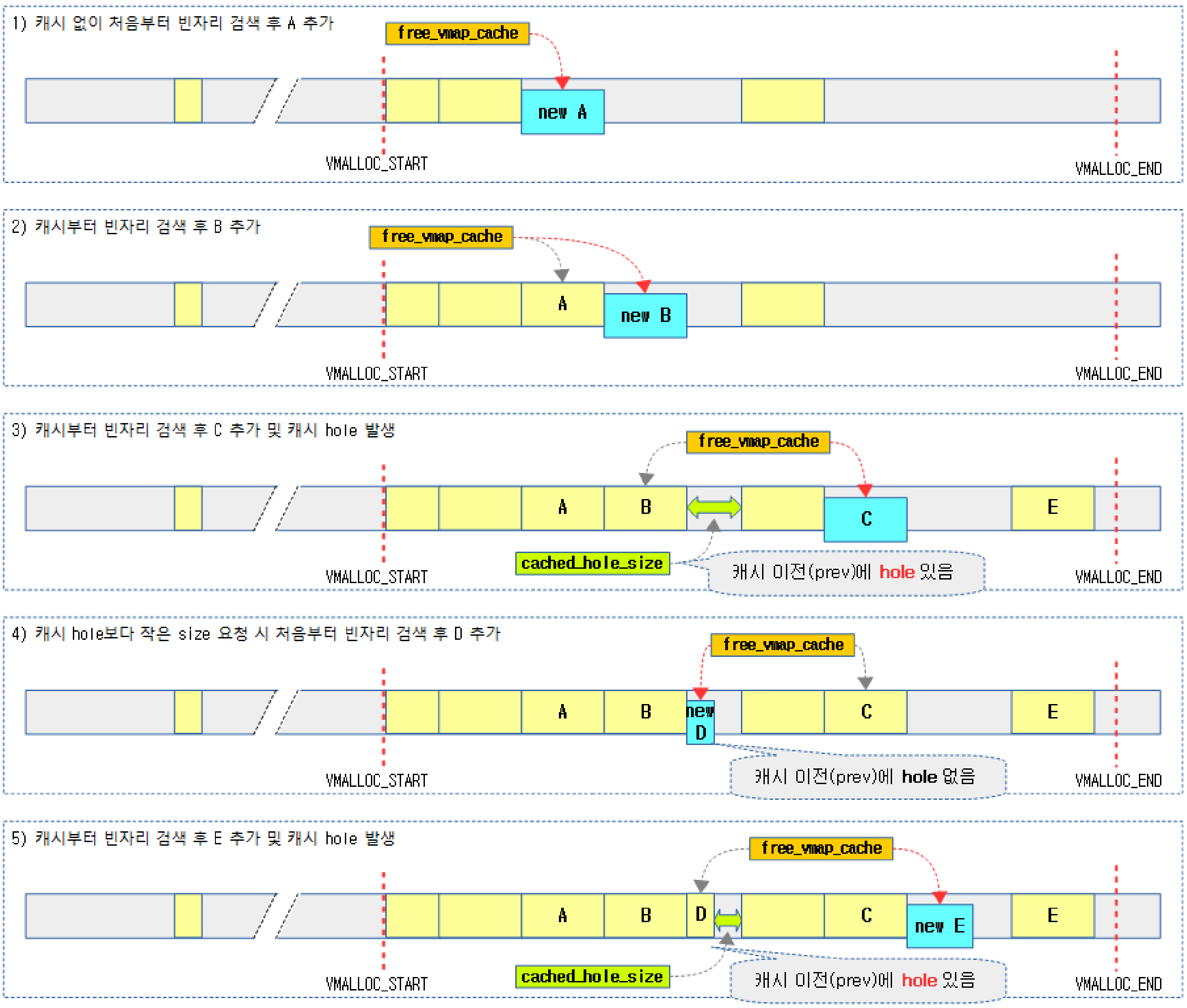
The following figure shows how when looking for an empty space within the request virtual address range for mapping, the first vmap_area is found in the RB tree, and the list is used again to search for the empty space for size.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_vmap_area-2b.png)

The following illustration shows the changes in free_vmap_cache and cached_hole_size.

- The cached_hole_size only remembers the size of the hole directly in front of the cache.


1) 캐시 없이 처음부터 빈자리 검색 후 A 추가
free_vmap_cache
new A
VMALLOC_START
VMALLOC_END
2) 캐시부터 빈자리 검색 후 B 추가
free_vmap_cache
A
new B
VMALLOC_START
VMALLOC_END
3) 캐시부터 빈자리 검색 후 C 추가 및 캐시 hole 발생
free_vmap_cache
A
B
C
E
cached_hole_size
캐시 이전(prev)에 hole 있음
VMALLOC_START
VMALLOC_END
4) 캐시 hole보다 작은 size 요청 시 처음부터 빈자리 검색 후 D 추가
free_vmap_cache
A
B
new D
C
E
캐시 이전(prev)에 hole 없음
VMALLOC_START
VMALLOC_END
5) 캐시부터 빈자리 검색 후 E 추가 및 캐시 hole 발생
free_vmap_cache
A
B
D
C
new E
E
cached_hole_size
캐시 이전(prev)에 hole 있음
VMALLOC_START
VMALLOC_END

(http://jake.dothome.co.kr/wp-content/uploads/2016/07/cached_hole_size-1.png)

Add vmap_area

__insert_vmap_area()

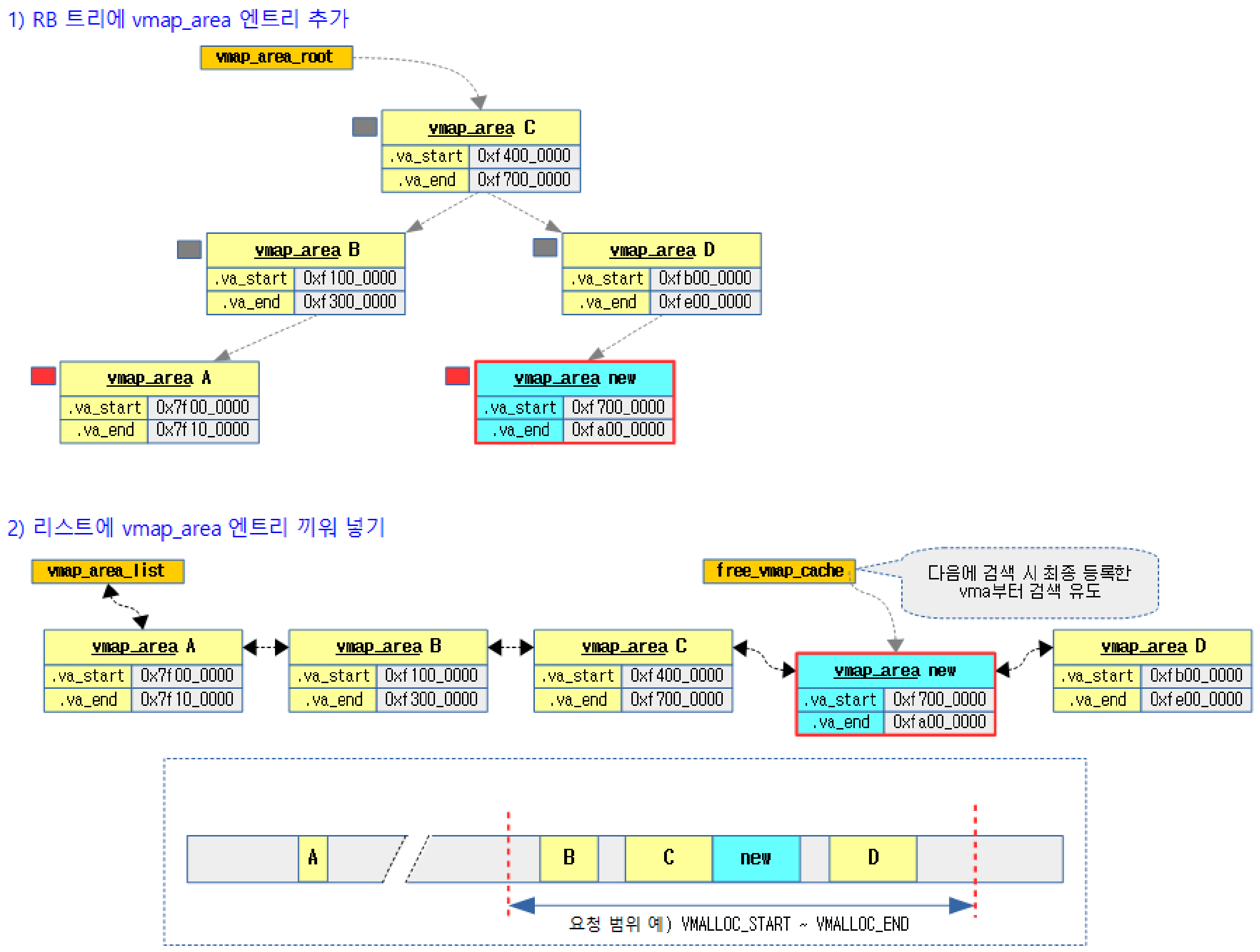
mm/vmalloc.c

```c
static void __insert_vmap_area(struct vmap_area *va)
{
        struct rb_node **p = &vmap_area_root.rb_node;
        struct rb_node *parent = NULL;
        struct rb_node *tmp;

        while (*p) {
                struct vmap_area *tmp_va;

                parent = *p;
                tmp_va = rb_entry(parent, struct vmap_area, rb_node);
                if (va->va_start < tmp_va->va_end)
                        p = &(*p)->rb_left;
                else if (va->va_end > tmp_va->va_start)
                        p = &(*p)->rb_right;
                else
                        BUG();
        }
19
20         rb_link_node(&va->rb_node, parent, p);
21         rb_insert_color(&va->rb_node, &vmap_area_root);
22
23         /* address-sort this list */
24         tmp = rb_prev(&va->rb_node);
25         if (tmp) {
26                 struct vmap_area *prev;
27                 prev = rb_entry(tmp, struct vmap_area, rb_node);
28                 list_add_rcu(&va->list, &prev->list);
29         } else
30                 list_add_rcu(&va->list, &vmap_area_list);
31 }
```

Insert vmap_area entries into the global vmap_area_root RB tree and global vmap_area_list.

- In code lines 7~18, find the leaf node you want to insert in the vmap_area_root RB tree.
- In line 20~21 of code, connect the entry to the leaf node and balance the RB tree.
- Finally, on lines 24~30 of code, use RCU to insert the vmap_area entry into the vmap_area_list.
    - If you use rb_prev() to insert an entry into the RB tree, you can find out which node is directly in front of it and insert it into the list connection.

The following illustration shows what it looks like when adding vmap_area entries.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/insert_vmap_area-1c.png)

vm_area Mapping

map_vm_area()

mm/vmalloc.c

```
int map_vm_area(struct vm_struct *area, pgprot_t prot, struct page **pages)
{
        unsigned long addr = (unsigned long)area->addr;
        unsigned long end = addr + get_vm_area_size(area);
        int err;

        err = vmap_page_range(addr, end, prot, pages);

        return err > 0 ? 0 : err;
}
EXPORT_SYMBOL_GPL(map_vm_area);
```

Map the requested vm_struct to the virtual address range contained in the information.

get_vm_area_size()

include/linux/vmalloc.h

```
static inline size_t get_vm_area_size(const struct vm_struct *area)
{
        if (!(area->flags & VM_NO_GUARD))
                /* return actual size without guard page */
                return area->size - PAGE_SIZE;
        else
                return area->size;

}
```

Returns the number of pages used by the zone.

vmap_page_range()

mm/vmalloc.c

```
static int vmap_page_range(unsigned long start, unsigned long end,
                           pgprot_t prot, struct page **pages)
{
        int ret;

        ret = vmap_page_range_noflush(start, end, prot, pages);
        flush_cache_vmap(start, end);
        return ret;
}
```

Map the requested virtual address range and flush that space.

vmap_page_range_noflush()

mm/vmalloc.c

```
/*
 * Set up page tables in kva (addr, end). The ptes shall have prot "prot", and
 * will have pfns corresponding to the "pages" array.
 *
 * Ie. pte at addr+N*PAGE_SIZE shall point to pfn corresponding to pages
[N]
 */
```

```
static int vmap_page_range_noflush(unsigned long start, unsigned long en
d,
                                   pgprot_t prot, struct page **pages)
{
        pgd_t *pgd;
        unsigned long next;
        unsigned long addr = start;
        int err = 0;
        int nr = 0;

        BUG_ON(addr >= end);
        pgd = pgd_offset_k(addr);
        do {
                next = pgd_addr_end(addr, end);
                err = vmap_p4d_range(pgd, addr, next, prot, pages, &nr);
                if (err)
                        return err;
        } while (pgd++, addr = next, addr != end);

        return nr;
}
```

Map the kernel page table corresponding to the request virtual address range using pages and attribute information.

- pgd -> p4d -> pud -> pmd -> pte table and map to the corresponding entries in the last pte table.

flush_cache_vmap() – ARM32

arch/arm/include/asm/cacheflush.h()

```
/*
 * flush_cache_vmap() is used when creating mappings (eg, via vmap,
 * vmalloc, ioremap etc) in kernel space for pages.  On non-VIPT
 * caches, since the direct-mappings of these pages may contain cached
 * data, we need to do a full cache flush to ensure that writebacks
 * don't corrupt data placed into these pages via the new mappings.
 */
static inline void flush_cache_vmap(unsigned long start, unsigned long e
nd)
{
        if (!cache_is_vipt_nonaliasing())
                flush_cache_all();
        else
                /*
                 * set_pte_at() called from vmap_pte_range() does not
                 * have a DSB after cleaning the cache line.
                 */
                dsb(ishst);
}
```

Flush on the requested virtual address range.

- If your architecture uses pipt cache or supports vipt nonaliasing, you don't need to flush, which greatly improves performance.

flush_cache_vmap() – ARM64

arch/arm64/include/asm/cacheflush.h

```
/*
 * Not required on AArch64 (PIPT or VIPT non-aliasing D-cache).
 */
```

```
static inline void flush_cache_vmap(unsigned long start, unsigned long end)
{
}
```

ARM64 does not flush the cache for vmap mapping.

vm_struct Settings

setup_vmalloc_vm()

mm/vmalloc.c

```
static void setup_vmalloc_vm(struct vm_struct *vm, struct vmap_area *va,
                              unsigned long flags, const void *caller)
{
        spin_lock(&vmap_area_lock);
        vm->flags = flags;
        vm->addr = (void *)va->va_start;
        vm->size = va->va_end - va->va_start;
        vm->caller = caller;
        va->vm = vm;
        va->flags |= VM_VM_AREA;
        spin_unlock(&vmap_area_lock);
}
```

Set up information in vm_struct and vmap_area.

Unmap vmap

vunmap()

mm/vmalloc.c

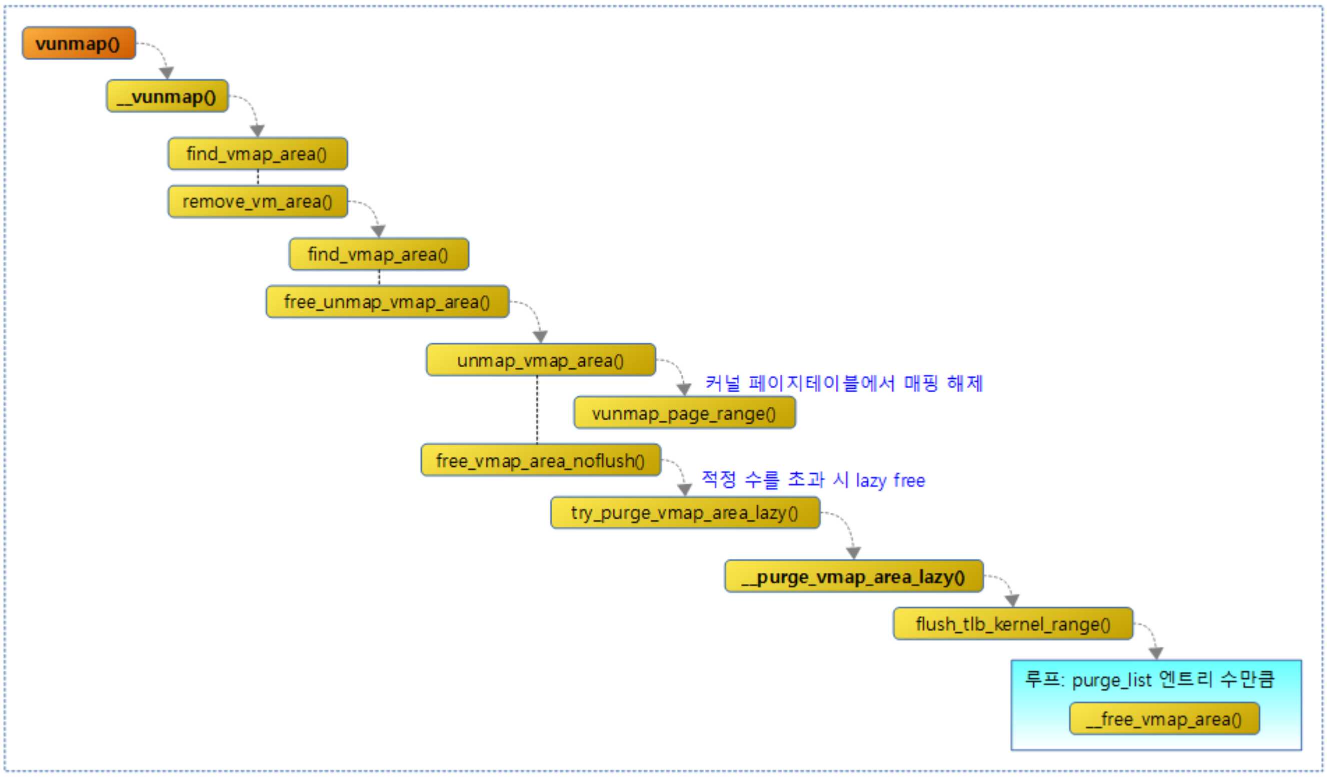
```
/**
 *      vunmap  -  release virtual mapping obtained by vmap()
 *      @addr:          memory base address
 *
 *      Free the virtually contiguous memory area starting at @addr,
 *      which was created from the page array passed to vmap().
 *
 *      Must not be called in interrupt context.
 */
```

```
void vunmap(const void *addr)
{
        BUG_ON(in_interrupt());
        might_sleep();
        if (addr)
                __vunmap(addr, 0);
}
```

```
EXPORT_SYMBOL(vunmap);
```

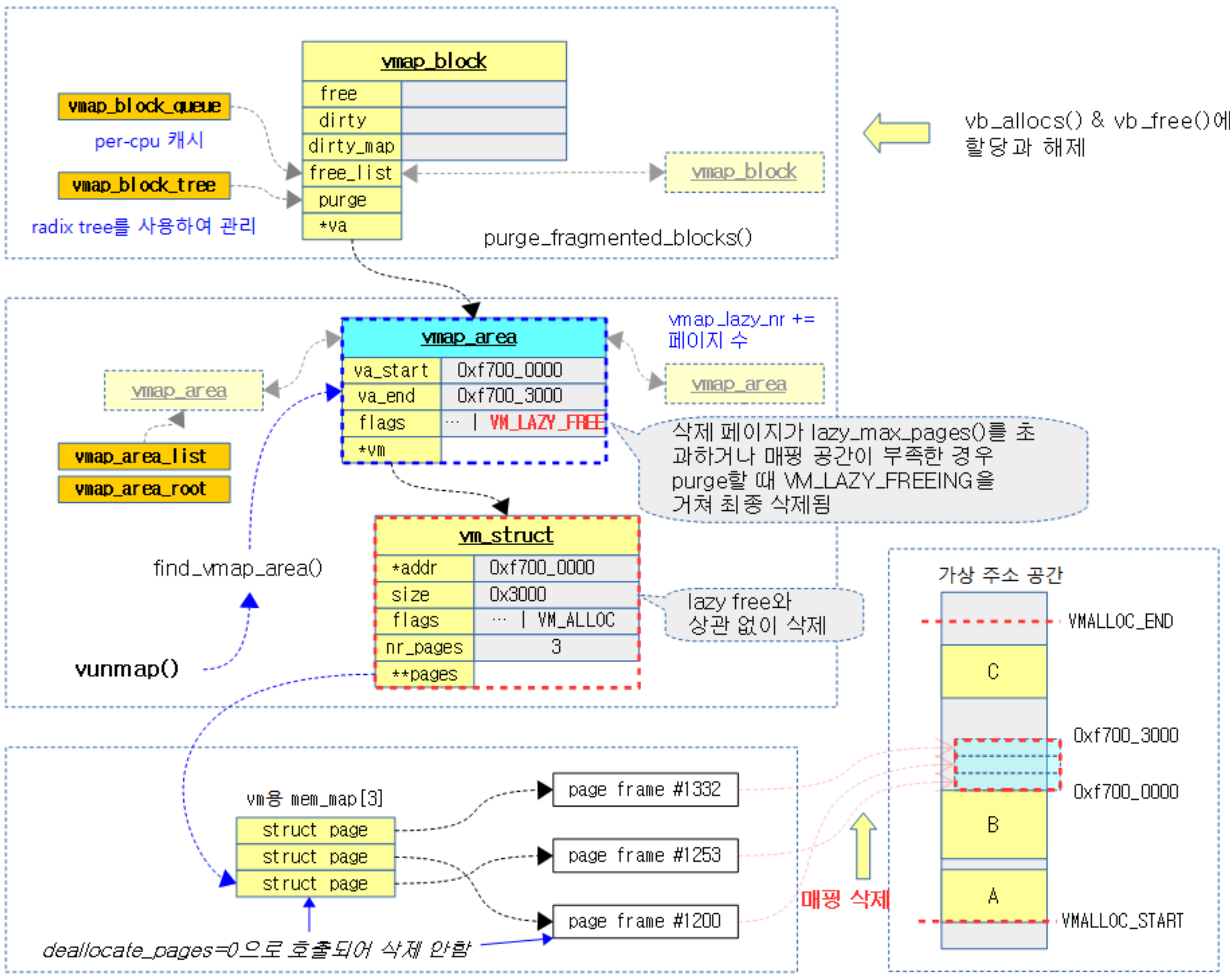
Use the vmap() function to unmap the virtual address zone mapped to the vmalloc address space. However, the physics page will not be deallocated.

The following figure shows the processing flow for the vunmap() function with its related functions.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vunmap-2a.png)

The following illustration shows the vummap() function finding the vmap_area() with the requested virtual address and discarding the corresponding mapping.



(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vunmap-1b.png)

__vunmap()

mm/vmalloc.c

```
static void __vunmap(const void *addr, int deallocate_pages)
{
        struct vm_struct *area;

        if (!addr)
                return;

        if (WARN(!PAGE_ALIGNED(addr), "Trying to vfree() bad address (%
p)\n",
                        addr))
                return;

        area = find_vmap_area((unsigned long)addr)->vm;
        if (unlikely(!area)) {
                WARN(1, KERN_ERR "Trying to vfree() nonexistent vm area
(%p)\n",
                                addr);
                return;
        }

        debug_check_no_locks_freed(area->addr, get_vm_area_size(area));
        debug_check_no_obj_freed(area->addr, get_vm_area_size(area));

        remove_vm_area(addr);
        if (deallocate_pages) {
```

```
24                  int i;
25
26                  for (i = 0; i < area->nr_pages; i++) {
27                          struct page *page = area->pages[i];
28
29                          BUG_ON(!page);
30                          __free_pages(page, 0);
31                  }
32
33                  kvfree(area->pages);
34          }
35
36          kfree(area);
37          return;
38  }
```

It finds the VM information with the request virtual address, removes the mapping, and returns the pages to the buddy system by releasing each page as requested.

- It retrieves the request virtual address from the vmap_area information registered in the RB tree vmap_area_root, removes and unmaps the matched vmap_area and vm_struct information, and frees the physical pages if the @deallocate_pages argument is requested.

- In lines 12~17 of the code, search for the VM by the request virtual address from the vmap_area information registered in the RB tree vmap_area_root.
- In line 22 of code, remove the VM from the RB tree and list, request unmapping, and return the VM.
- If a @deallocate_pages argument request is set in code lines 23~34, all registered pages will be released and returned to the buddy system.
- Unassign the vm_area struct information from lines 36~37 of code.

vmap_area Deletion and Unmapping Requests

remove_vm_area()

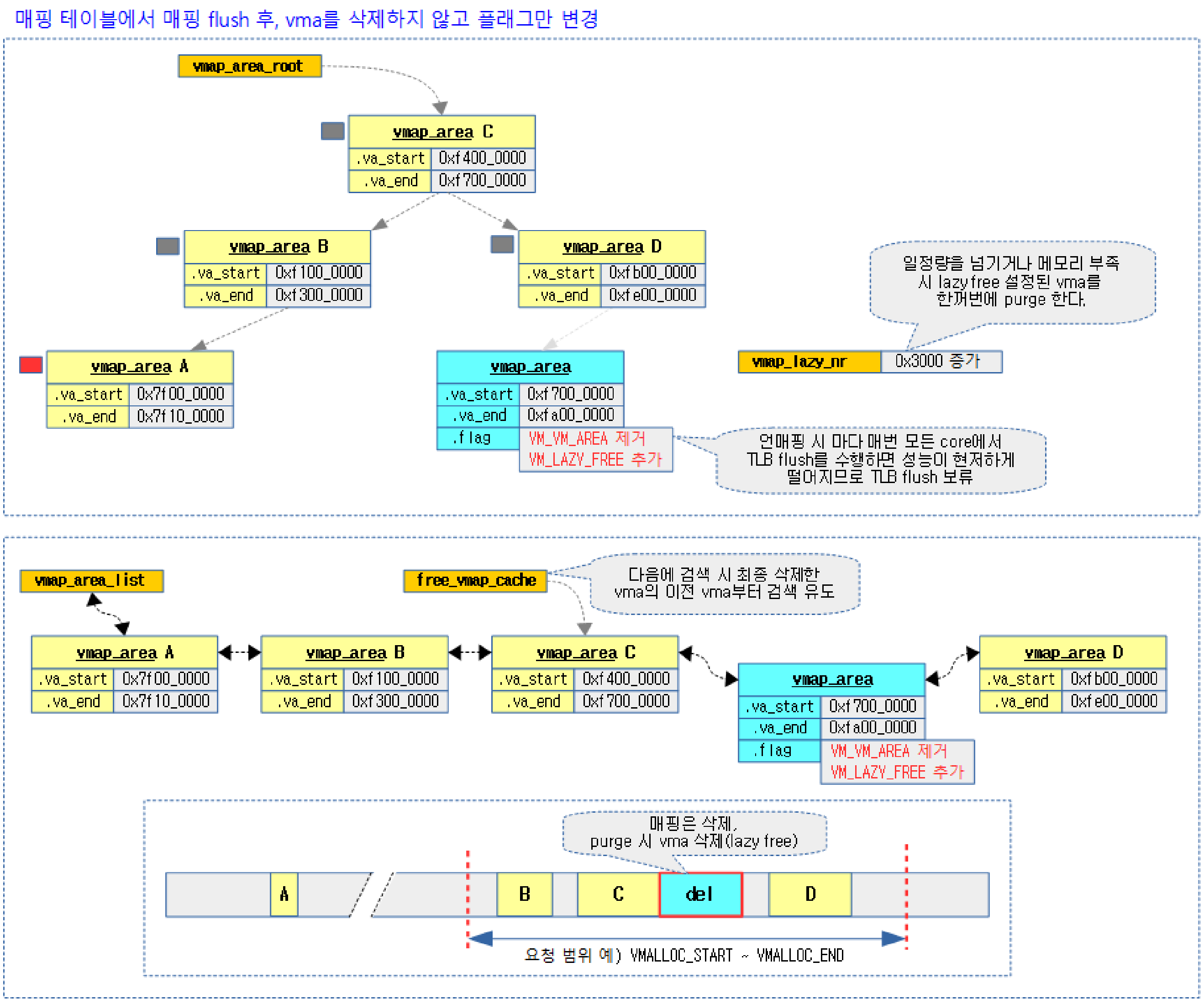
mm/vmalloc.c

```
/**
 *      remove_vm_area  -  find and remove a continuous kernel virtual area
 *      @addr:          base address
 *
 *      Search for the kernel VM area starting at @addr, and remove it.
 *      This function returns the found VM area, but using it is NOT safe
 *      on SMP machines, except for its size or flags.
 */
```

```
struct vm_struct *remove_vm_area(const void *addr)
{
        struct vmap_area *va;

        might_sleep();

        va = find_vmap_area((unsigned long)addr);
        if (va && va->flags & VM_VM_AREA) {
                struct vm_struct *vm = va->vm;
                spin_lock(&vmap_area_lock);
                va->vm = NULL;
                va->flags &= ~VM_VM_AREA;
                va->flags |= VM_LAZY_FREE;
                spin_unlock(&vmap_area_lock);

                kasan_free_shadow(vm);
                free_unmap_vmap_area(va);

                return vm;
        }
        return NULL;
}
```

The request virtual address is retrieved from the vmap_area information registered in the RB tree vmap_area_root, the matched vmap_area information is removed, the unmap is requested, and the vm_struct information is obtained.

- Memory mapping is turned off, but vm_area simply deletes the VM_VM_AREA flag and adds the VM_LAZY_FREE flag.
- In fact, when the number of vmap_lazy_nr exceeds a certain amount or there is a shortage of memory, the VM_LAZY_FREE VMas are purge all at once.
  - If the mapping of the vmalloc address space is revoked or modified, a TLB flush must occur on all cores, which is significantly degraded if this is done every time, so the method is to collect the items to be deleted and delete them all at once. This is called Lazy TLB flushing.


매핑 테이블에서 매핑 flush 후, vma를 삭제하지 않고 플래그만 변경
vmap_area_root
vmap_area C
.va_start 0xf400_0000
.va_end 0xf700_0000
vmap_area B
.va_start 0xf100_0000
.va_end 0xf300_0000
vmap_area D
.va_start 0xfb00_0000
.va_end 0xfe00_0000
vmap_area A
.va_start 0x7f00_0000
.va_end 0x7f10_0000
vmap_area
.va_start 0xf700_0000
.va_end 0xfa00_0000
.flag VM_VM_AREA 제거 VM_LAZY_FREE 추가
일정량을 넘기거나 메모리 부족 시 lazy free 설정된 vma를 한꺼번에 purge 한다.
vmap_lazy_nr 0x3000 증가
언매핑 시 마다 매번 모든 core에서 TLB flush를 수행하면 성능이 현저하게 떨어지므로 TLB flush 보류
vmap_area_list
free_vmap_cache
다음에 검색 시 최종 삭제한 vma의 이전 vma부터 검색 유도
매핑은 삭제, purge 시 vma 삭제(lazy free)
A
B
C
del
D
요청 범위 예) VMALLOC_START ~ VMALLOC_END

(http://jake.dothome.co.kr/wp-content/uploads/2016/07/remove_vm_area-1.png)

vmap_area Search

find_vmap_area()

mm/vmalloc.c

```
static struct vmap_area *find_vmap_area(unsigned long addr)
{
        struct vmap_area *va;

        spin_lock(&vmap_area_lock);
        va = __find_vmap_area(addr);
        spin_unlock(&vmap_area_lock);

        return va;
}
```

After being protected by a vmap_area lock, the vmap_area information is retrieved from the request virtual address. If not found, it returns null.

__find_vmap_area()

mm/vmalloc.c

```
static struct vmap_area *__find_vmap_area(unsigned long addr)
{
        struct rb_node *n = vmap_area_root.rb_node;

        while (n) {
                struct vmap_area *va;

                va = rb_entry(n, struct vmap_area, rb_node);
                if (addr < va->va_start)
                        n = n->rb_left;
                else if (addr >= va->va_end)
                        n = n->rb_right;
                else
                        return va;
        }

        return NULL;
}
```

Retrieve vmap_area information from the request virtual address. If not found, it returns null.

- Search for the request virtual address from the vmap_area information registered in the RB tree vmap_area_root to find the matched vmap_area.

Unmap vmap_area

free_unmap_vmap_area()

mm/vmalloc.c

```
/*
 * Free and unmap a vmap area
 */
```

```
static void free_unmap_vmap_area(struct vmap_area *va)
{
        flush_cache_vunmap(va->va_start, va->va_end);
        unmap_vmap_area(va);
        if (debug_pagealloc_enabled())
                flush_tlb_kernel_range(va->va_start, va->va_end);

        free_vmap_area_noflush(va);
}
```

Depending on the architecture, the data cache for the specified virtual address range is emptied, and the area is unmapped.

flush_cache_vunmap() – ARM32

arch/arm/include/asm/cacheflush.h

```
static inline void flush_cache_vunmap(unsigned long start, unsigned long end)
{
        if (!cache_is_vipt_nonaliasing())
                flush_cache_all();
}
```

If the architecture's data cache is of type vivt or vipt aliasing, empty the cache altogether.

- If the data cache is of type pipt or vipt nonaliasing, there is no need to empty the cache, which improves performance.

flush_cache_vunmap() – ARM64

arch/arm64/include/asm/cacheflush.h

```
static inline void flush_cache_vunmap(unsigned long start, unsigned long end)
{
}
```

ARM64 does not flush the cache for vmap demapping.

free_unmap_vmap_area_noflush()

mm/vmalloc.c

```
/*
 * Free and unmap a vmap area, caller ensuring flush_cache_vunmap had been
 * called for the correct range previously.
 */
```

```
static void free_unmap_vmap_area_noflush(struct vmap_area *va)
{
        unmap_vmap_area(va);
        free_vmap_area_noflush(va);
}
```

Unmap the virtual address area used by the vmap_area from the kernel page table. Then, instead of deleting the vmap_area right away, add it to the purge_list.

- When vmap_area is recycled, performance is greatly improved because there is no need to flush the time-consuming TLB cache.

Unmap a page table

unmap_vmap_area()

mm/vmalloc.c

```
/*
 * Clear the pagetable entries of a given vmap_area
 */
```

```
static void unmap_vmap_area(struct vmap_area *va)
{
        vunmap_page_range(va->va_start, va->va_end);
}
```

Disable the mapping of the virtual address area of the vmap_area from the kernel page table.

vunmap_page_range()

mm/vmalloc.c

```
static void vunmap_page_range(unsigned long addr, unsigned long end)
{
        pgd_t *pgd;
        unsigned long next;

        BUG_ON(addr >= end);
        pgd = pgd_offset_k(addr);
        do {
                next = pgd_addr_end(addr, end);
                if (pgd_none_or_clear_bad(pgd))
                        continue;
                vunmap_p4d_range(pgd, addr, next);
        } while (pgd++, addr = next, addr != end);
}
```

Unmap the request virtual address range from the kernel page table.

- If you go to the pgd -> p4d -> pud -> pmd -> pte table, it will unmap the corresponding entries in the last pte table.

Request lazy TLB Flush

free_vmap_area_noflush()

mm/vmalloc.c

```
/*
 * Free a vmap area, caller ensuring that the area has been unmapped
 * and flush_cache_vunmap had been called for the correct range
 * previously.
 */
```

```
static void free_vmap_area_noflush(struct vmap_area *va)
{
        int nr_lazy;

        nr_lazy = atomic_add_return((va->va_end - va->va_start) >> PAGE_SHIFT,
                                    &vmap_lazy_nr);

        /* After this point, we may free va at any time */
        llist_add(&va->purge_list, &vmap_purge_list);

        if (unlikely(nr_lazy > lazy_max_pages()))
                try_purge_vmap_area_lazy();
}
```

Instead of deleting the vmap_area right away, it sets a VM_LAZY_FREE flag to request a lazy TLB flush. Then, if the number of pages exceeds a certain amount, a purge processing is performed.

lazy_max_pages()

mm/vmalloc.c

```
/*
 * lazy_max_pages is the maximum amount of virtual address space we gather up
 * before attempting to purge with a TLB flush.
 *
 * There is a tradeoff here: a larger number will cover more kernel page tables
 * and take slightly longer to purge, but it will linearly reduce the number of
 * global TLB flushes that must be performed. It would seem natural to scale
 * this number up linearly with the number of CPUs (because vmapping activity
 * could also scale linearly with the number of CPUs), however it is likely
 * that in practice, workloads might be constrained in other ways that mean
 * vmap activity will not scale linearly with CPUs. Also, I want to be
 * conservative and not introduce a big latency on huge systems, so go with
 * a less aggressive log scale. It will still be an improvement over the old
 * code, and it will be simple to change the scale factor if we find that it
 * becomes a problem on bigger systems.
 */
```

```
static unsigned long lazy_max_pages(void)
{
        unsigned int log;

        log = fls(num_online_cpus());

        return log * (32UL * 1024 * 1024 / PAGE_SIZE);
}
```

Calculate the number of pages required for purge processing of TLB lazy pages.

- Returns the number of pages equal to 32M * log2(online cpu) + 1
- As the number of CPUs increases, TLB flush degrades the overall performance of the system, so the higher the number of CPUs, the greater the number of lazy_max_pages.

Purge processing for lazy TLB vmap_area

try_purge_vmap_area_lazy()

mm/vmalloc.c

```
/*
 * Kick off a purge of the outstanding lazy areas. Don't bother if somebody
 * is already purging.
 */
```

```
static void try_purge_vmap_area_lazy(void)
{
        if (mutex_trylock(&vmap_purge_lock)) {
                __purge_vmap_area_lazy(ULONG_MAX, 0);
                mutex_unlock(&vmap_purge_lock);
        }
}
```

Lazy TLB Flush removes the requested vmap_area and performs a TLB flush of the entire virtual address range.

__purge_vmap_area_lazy()

mm/vmalloc.c

```
/*
 * Purges all lazily-freed vmap areas.
 */
```

```
01 static bool __purge_vmap_area_lazy(unsigned long start, unsigned long end)
02 {
03         struct llist_node *valist;
04         struct vmap_area *va;
05         struct vmap_area *n_va;
06         bool do_free = false;
07 
08         lockdep_assert_held(&vmap_purge_lock);
09 
10         valist = llist_del_all(&vmap_purge_list);
11         llist_for_each_entry(va, valist, purge_list) {
12                 if (va->va_start < start)
13                         start = va->va_start;
14                 if (va->va_end > end)
15                         end = va->va_end;
16                 do_free = true;
17         }
18 
19         if (!do_free)
20                 return false;
21 
22         flush_tlb_kernel_range(start, end);
23 
24         spin_lock(&vmap_area_lock);
25         llist_for_each_entry_safe(va, n_va, valist, purge_list) {
26                 int nr = (va->va_end - va->va_start) >> PAGE_SHIFT;
27 
28                 __free_vmap_area(va);
29                 atomic_sub(nr, &vmap_lazy_nr);
30                 cond_resched_lock(&vmap_area_lock);
31         }
32         spin_unlock(&vmap_area_lock);
33         return true;
34 }
```

Lazy TLB Flush Remove all requested vmap_area and perform a TLB flush in the range of @start~@end.

- In code lines 10~17, update the start and end addresses to be flushed to reflect the first and last end addresses of the vmap_area in the purge list.
- Perform a TLB flush for the virtual address zone in the range start ~ end that was updated on line 22 of the code.
- In code lines 24~32, deallocate all vmap_area in the purge list.
- On line 33 of the code, it returns true for success.

Delete vmap_area

__free_vmap_area()

mm/vmalloc.c

```
01 static void __free_vmap_area(struct vmap_area *va)
02 {
03         BUG_ON(RB_EMPTY_NODE(&va->rb_node));
```

```

        if (free_vmap_cache) {
                if (va->va_end < cached_vstart) {
                        free_vmap_cache = NULL;
                } else {
                        struct vmap_area *cache;
                        cache = rb_entry(free_vmap_cache, struct vmap_area, rb_node);
                        if (va->va_start <= cache->va_start) {
                                free_vmap_cache = rb_prev(&va->rb_node);
                                /*
                                 * We don't try to update cached_hole_size or
                                 * cached_align, but it won't go very wrong.
                                 */
                        }
                }
        }
        rb_erase(&va->rb_node, &vmap_area_root);
        RB_CLEAR_NODE(&va->rb_node);
        list_del_rcu(&va->list);

        /*
         * Track the highest possible candidate for pcpu area
         * allocation.  Areas outside of vmalloc area can be returned
         * here too, consider only end addresses which fall inside
         * vmalloc area proper.
         */
        if (va->va_end > VMALLOC_START && va->va_end <= VMALLOC_END)
                vmap_area_pcpu_hole = max(vmap_area_pcpu_hole, va->va_end);

        kfree_rcu(va, rcu_head);
}
```

Remove the requested vmap_area from the RB tree vmap_area_root and vmap_area_list and release it.

consultation

- Kmalloc vs Vmalloc (http://jake.dothome.co.kr/kmalloc-vs-vmalloc) | 문c
- Kmalloc (http://jake.dothome.co.kr/kmalloc) | Qc
- Vmalloc (http://jake.dothome.co.kr/vmalloc) | Qc
- Vmap((http://jake.dothome.co.kr/vmap)) | Sentence C – Current post

- Red–black tree (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree) | wikipedia
- Reworking vmap() (https://lwn.net/Articles/304188/) | LWN.net
- mm: rewrite vmap layer (https://github.com/torvalds/linux/commit/db64fe02258f1507e13fe5212a989922323685ce)
- mm:rewrite vmap layer (http://barriosstory.blogspot.kr/2009/01/mmrewrite-vmap-layer.html) | Barrios

12 thoughts to “Vmap”

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2017-02-07 11:36 (http://jake.dothome.co.kr/vmap/#comment-125380)

Upon re-analysis, the free_vmap_cache interpretation was incorrect and corrected.

RESPONSE (/VMAP/?REPLYTOCOM=125380#RESPOND)

GARDENING
2019-08-29 13:45 (http://jake.dothome.co.kr/vmap/#comment-216600)

Thank you for posting the informative material~
Regarding
vmap_area allocation, if (!free_vmap_cache ||size < cached_hole_size ||vstart < cached_vstart ||align < cached_align) {
Why can't the cache be used if the requested size is less than cached_hole_size? Rather, the cached_hole_size has to be smaller than the size to be able to use that space, so shouldn't it disable the cache? Other vstarts and aligns seem to be the same for me, can you explain?

RESPONSE (/VMAP/?REPLYTOCOM=216600#RESPOND)

GARDENING
2019-08-29 18:37 (http://jake.dothome.co.kr/vmap/#comment-216614)

If the
size is smaller than cached_hole_size, I guess that's because I wanted to search for it from the beginning.

RESPONSE (/VMAP/?REPLYTOCOM=216614#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-08-30 08:54 (http://jake.dothome.co.kr/vmap/#comment-216638)

Yes, you found out the reason for my late response. ^^;

If you can't get the space you need to use cached_hole_size to make a quick decision, then you use the regular method,
which uses the rb tree and list to do a range search.

I appreciate it.

RESPONSE (/VMAP/?REPLYTOCOM=216638#RESPOND)

WONHYUK YANG (HTTPS://WWW.BHRAL.COM/)
2022-01-15 00:54 (http://jake.dothome.co.kr/vmap/#comment-306237)

Hello, this is Yang Won-hyuk from the 16th generation. Thank you for the helpful article as always.

Apparently I found a typo, so I'm leaving a comment.
In the attached code in free_unmap_vmap_area(), it calls a free_unmap_vmap_area_noflush function, but from looking at the v5.0 tag, it seems that it should be fixed to a free_vmap_area_noflush function call,
not a free_unmap_vmap_area_noflush function call.

I was looking at the article and thinking "Why does the unmap_vmap_area function call twice?" and I found it ^^;

RESPONSE (/VMAP/?REPLYTOCOM=306237#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2022-01-15 14:39 (http://jake.dothome.co.kr/vmap/#comment-306239)

안녕하세요? 양원혁님!

커널 v4.9 코드를 v5.0으로 변환하는 과정에서 해당 라인이 수정되지 않았었군요.
말씀해주신대로 수정하였습니다. (v4.9때에는 free_unmap_vmap_area_noflush() 함수를 호출했었거든요 ^^;)

감사합니다. 좋은 하루 되세요.

응답 (/VMAP/?REPLYTOCOM=306239#RESPOND)

다로
2022-03-25 22:18 (http://jake.dothome.co.kr/vmap/#comment-306436)

안녕하세요

alloc_vmap_area() 함수와 그림 설명이 일치하지 않는 것 같아 문의드립니다.
free_vmap_cache == null이면, RB 트리의 root 노드부터 찾아서 내려가는 것으로 이해를 했습니다. while(n) { } 문을 벗어나는 조건은 if(tmp->va_start va_start <= addr)가 참이 되지 못해서 tmp가 왼쪽 노드로 이동하는 것으로 생각되는데... 혹시 제가 잘못 생각하는 것 일까요?

응답 (/VMAP/?REPLYTOCOM=306436#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2022-03-26 14:58 (http://jake.dothome.co.kr/vmap/#comment-306439)

```
while (n) {
struct vmap_area *tmp;
tmp = rb_entry(n, struct vmap_area, rb_node);
if (tmp->va_end >= addr) {
first = tmp;
if (tmp->va_start < = addr)
break;
n = n->rb_left;
} else
n = n->rb_right;
}
```

트리 검색시 addr가 좌측 하위 vma와 부모 vma 사이에 있을 때 좌측 하위의 vma가 first로 지정됩니다.

위의 코드와 관련된 그림이라면 http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_vmap_area-2b.png (http://jake.dothome.co.kr/wp-content/uploads/2016/07/alloc_vmap_area-2b.png) 인데, 어느 부분이 잘못되었다고 생각하세요?

응답 (/VMAP/?REPLYTOCOM=306439#RESPOND)

다로
2022-03-26 20:21 (http://jake.dothome.co.kr/vmap/#comment-306442)

아 네 말씀하신 그림에서 first가 vmap_area A가 되어야 하는 게 아닌가 생각했습니다
addr = ALIGN(VMALLOC_START, align)이고, 좌측하위 vmap_area A와 부모 vmap_area B 사이에 있는 것으로 되어 있어서..
(위 내용도 댓글에 썼었는데,, 올리다가 제가 실수로 지운것 같습니다..;;)

응답 (/VMAP/?REPLYTOCOM=306442#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2022-03-26 23:17 (http://jake.dothome.co.kr/vmap/#comment-306444)

순간적으로 제가 답변을 오해할 수 있게 작성하였군요.

다시 말씀드리겠습니다.

인자로 전달받은 @vstart(vmalloc 공간의 시작 주소 등)를 align 단위로 정렬하여 addr를 얻어냅니다.
그 후 이 addr와 RB 트리를 루트노드부터 비교하여 해당 범위에 들어가는 first 노드를 찾아냅니다.

예를 들어 vmalloc 공간의 시작위치 @vstart를 0xf000_0000 이라고 하겠습니다.
while 문을 통해 매 루트 노드 C부터 시작하여 if (tmp->va_end >= addr) 조건을 진행하면,
루트 노드인 C와 그 다음 좌측 B노드에 대해서는 만족하게 됩니다.
그런데 3번째 A 노드를 비교할 때에는 A노드가 vmalloc 공간 범위를 벗어난 좌측에 있으므로 false가 되며,
n = n->rb_right를 받아서 null이 대입되므로 이전까지 성공했던 B 노드를 선택하게되어 있습니다.

그 후부터는 리스트를 통해 빈공간을 검색합니다. 그림 표현은 수준이 낮지만 이상한 점은 발견되지 않고 있습니다 ^^;

감사합니다.

응답 (/VMAP/?REPLYTOCOM=306444#RESPOND)

다로
2022-03-27 18:06 (http://jake.dothome.co.kr/vmap/#comment-306445)

답변 감사합니다
제가 코드를 좀 더 주의깊게 봤어야 했는데 순간적으로 헷갈렸습니다.

자세하게 답변해주셔서 감사합니다!

응답 (/VMAP/?REPLYTOCOM=306445#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2022-03-28 15:22 (http://jake.dothome.co.kr/vmap/#comment-306450)

별말씀을요. 오늘도 좋은 하루되세요. ^^

응답 (/VMAP/?REPLYTOCOM=306450#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

Vmalloc (http://jake.dothome.co.kr/vmalloc/)

IDR(integer ID 관리) (http://jake.dothome.co.kr/idr/)