Vmalloc

2016-07-21 (http://jake.dothome.co.kr/vmalloc/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Vmalloc Allocator

The kernel is mainly allocated and used a large contiguous virtual address space. Since it is a non-physically contiguous space, it cannot be used as a DMA buffer.

feature

It is used when a large amount of kernel memory is required to be allocated with a contiguous virtual address. The allocation of kernel memory using vmalloc has the following characteristics:

- Every time memory is allocated, it can be mapped to the vmalloc virtual address space, which is expensive to map/decommission.
- Memory pages that map to the vmalloc virtual address space are well suited for fragment management, as they are allocated and used only as many single pages with order 0 as needed by the page allocator using the buddy system.
- It does not use physically contiguous memory, so it cannot be used for DMA.
  - Note: If you are using an IOMMU device for dynamic mapping, there is a way for the device to use this area as a DMA buffer for memory that is not physically contiguous. Currently, devices such as GPUs in mobile systems use ARM's SMMU (IOMMU), and the number is expected to increase.

API

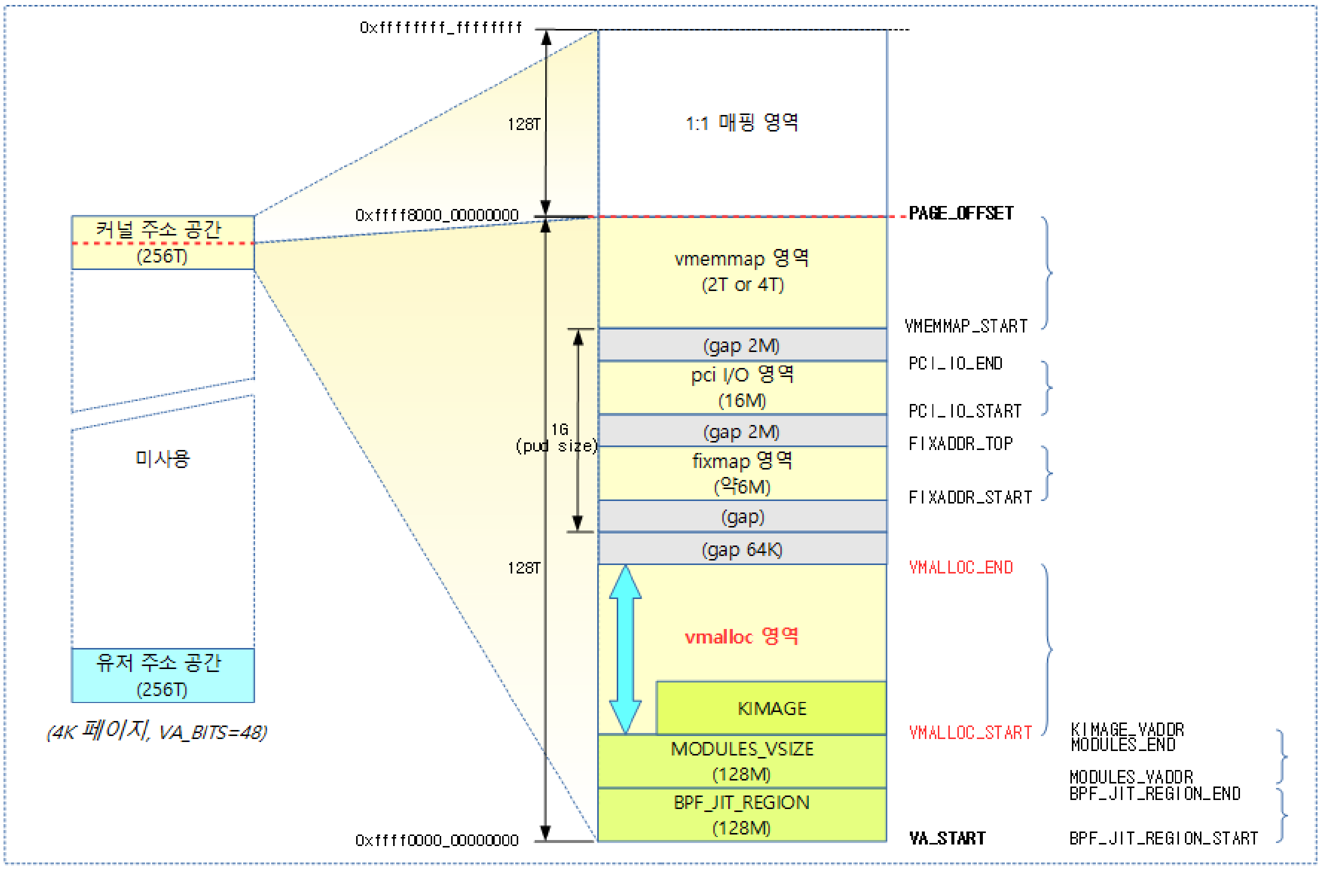
The main APIs for allocation and release are as follows:

- vmalloc()
- vfree()

vmalloc virtual address space

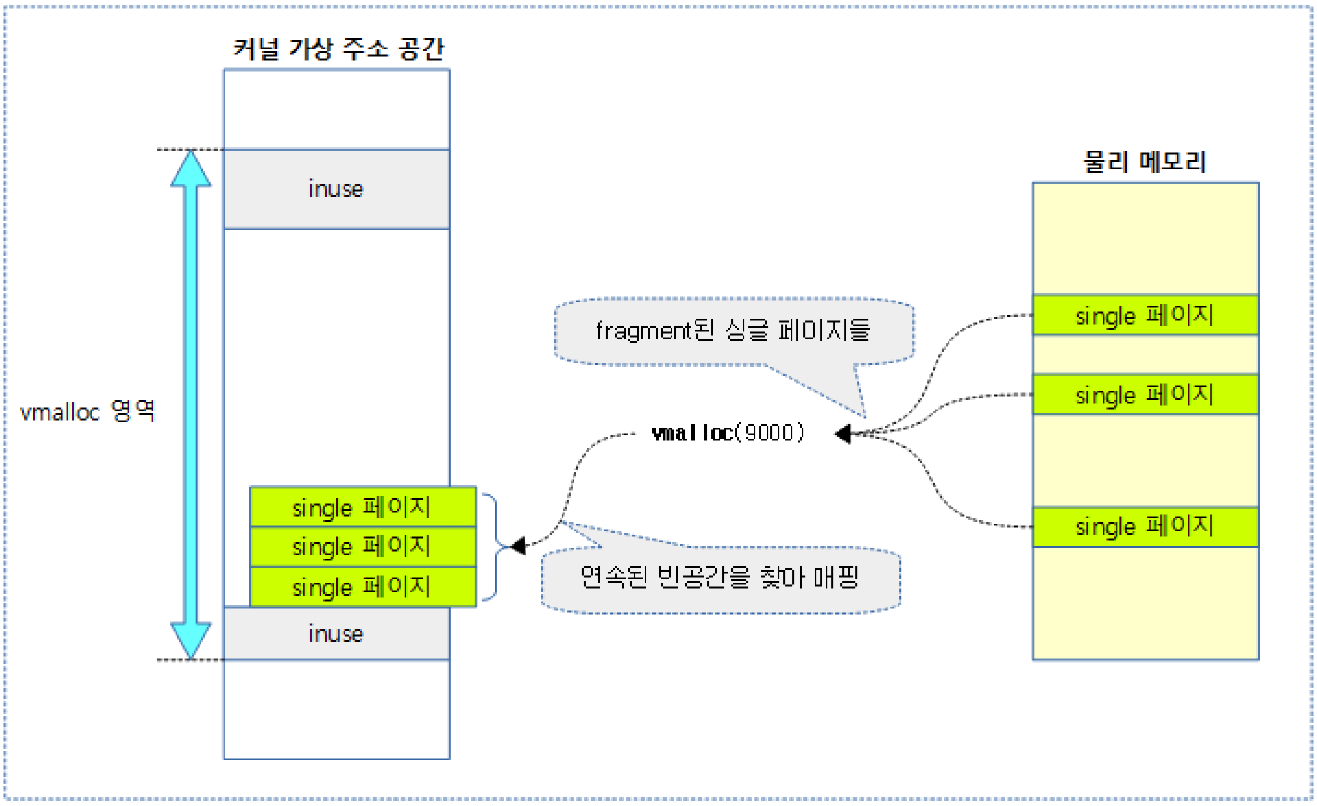
The page allocator using the buddy system receives a large number of single pages that can fit the size requested by the vmalloc() function, collects them, finds an empty space in the vmalloc virtual address space, maps each single page sequentially to the empty place, and returns the mapped virtual address.

The following figure shows the location of the vmalloc virtual address space.



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

The vmalloc() function takes a bunch of single pages that can fit the requested size, finds the empty spots in the vmalloc virtual address area, and maps them all together in a contiguous manner.



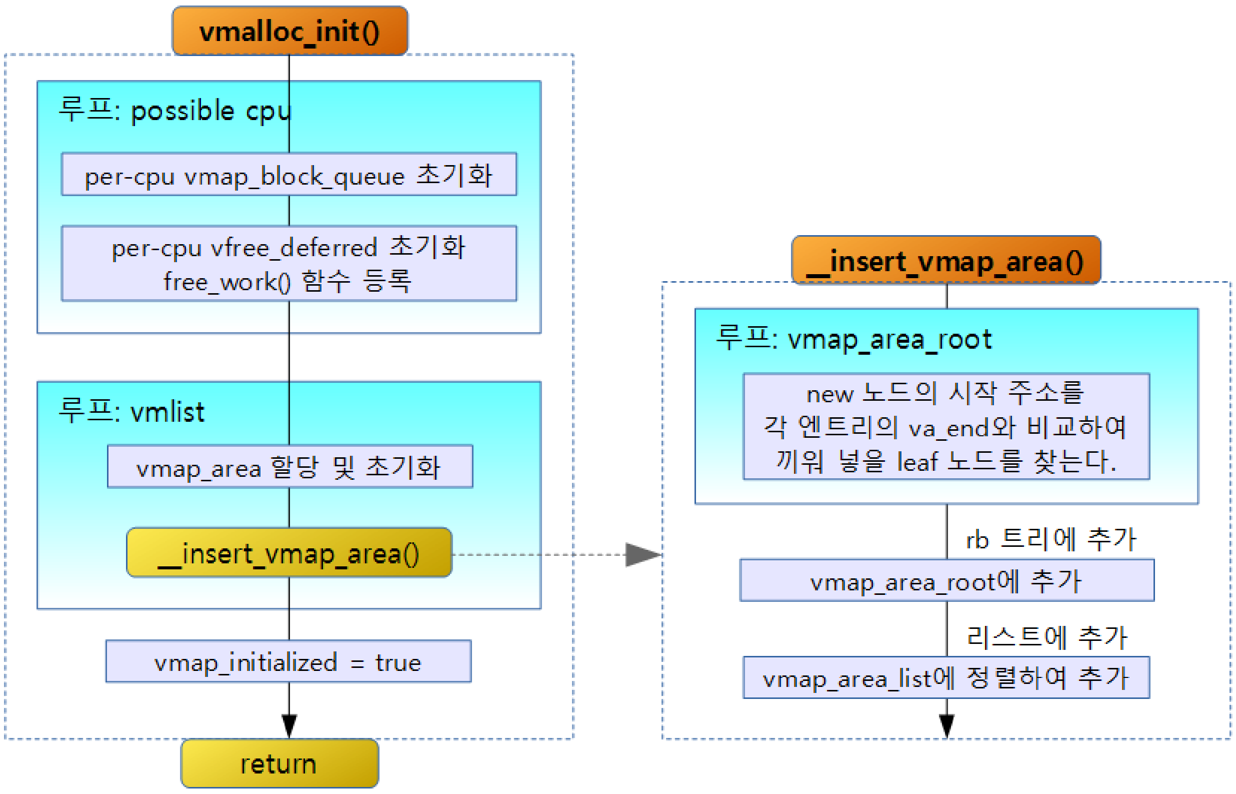
(http://jake.dothome.co.kr/wp-content/uploads/2016/07/vmalloc-3.png)

Initialize vmalloc

Initialization is performed for the mechanism of allocating contiguous virtual address memory used by the kernel.

- vmalloc() & vfree()
  - Contiguous virtual address memory allocation and memory freeing
- vmap() & vunmap()
  - Mapping and unmapping pages to a virtual address space
    - There are currently two virtual address spaces available: the vmalloc virtual address space and the module virtual address space.

The following figure shows the processing flow of the vmalloc_init() function.



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

vmalloc_init()

mm/vmalloc.c

```
void __init vmalloc_init(void)
{
        struct vmap_area *va;
        struct vm_struct *tmp;
        int i;

        for_each_possible_cpu(i) {
                struct vmap_block_queue *vbq;
09                  struct vfree_deferred *p;
10 
11                  vbq = &per_cpu(vmap_block_queue, i);
12                  spin_lock_init(&vbq->lock);
13                  INIT_LIST_HEAD(&vbq->free);
14                  p = &per_cpu(vfree_deferred, i);
15                  init_llist_head(&p->list);
16                  INIT_WORK(&p->wq, free_work);
17          }
18 
19          /* Import existing vmlist entries. */
20          for (tmp = vmlist; tmp; tmp = tmp->next) {
21                  va = kzalloc(sizeof(struct vmap_area), GFP_NOWAIT);
22                  va->flags = VM_VM_AREA;
23                  va->va_start = (unsigned long)tmp->addr;
24                  va->va_end = va->va_start + tmp->size;
25                  va->vm = tmp;
26                  __insert_vmap_area(va);
27          }
28 
29          vmap_area_pcpu_hole = VMALLOC_END;
30 
31          vmap_initialized = true;
32  }
```

Initialize the vmalloc space to be available.

- In line 7~17 of the code, loop around the number of possible CPUs and initialize the per-cpu vmap_block_queue and per-cpu vfree_deferred.
    - per-cpu vmap_block_queue
        - It is used as a free and dirty vmap block queue for allocation and flushing purposes.
    - per-cpu vfree_deferred
        - Wait risk used for the purpose of delaying vfree() processing during interrupt processing
- In line 20~27 of the code, vmap_area structs are allocated for the number registered in the global vmlist, initialized and added to the vmap_area.
    - vmlist
        - The VM is registered via the vm_area_register_early() function, which is used to create the pcpu_page_first_chunk().
            - It is also used when called from the xen ops driver on x86.
- In lines 29~31 of the code, the per-cpu will be used from the top of the vmalloc space to the bottom. Announce that vmap is ready to use.

---

vmalloc allocation

vmalloc()

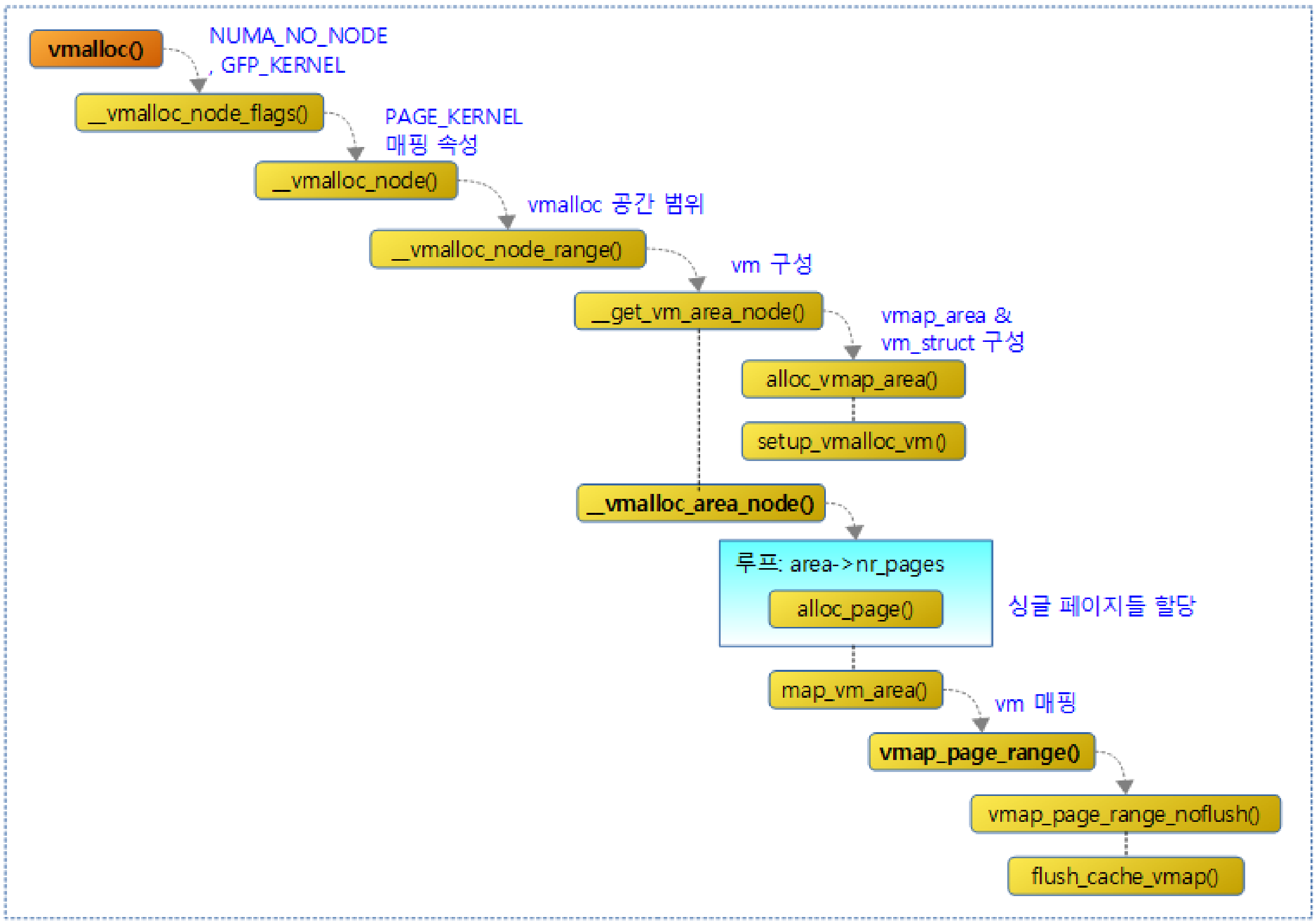
mm/vmalloc.c

```
/**
 *      vmalloc  -  allocate virtually contiguous memory
 *      @size:          allocation size
 *      Allocate enough pages to cover @size from the page level
 *      allocator and map them into contiguous kernel virtual space.
 *
 *      For tight control over page level allocator and protection flags
 *      use __vmalloc() instead.
 */
```

```
void *vmalloc(unsigned long size)
{
        return __vmalloc_node_flags(size, NUMA_NO_NODE,
                                    GFP_KERNEL);
}
EXPORT_SYMBOL(vmalloc);
```

It is allocated a @size contiguous virtual address space for the kernel. __vmalloc_node_flags() function.

The following illustration shows the processing flow between the vmalloc() function and its related functions.



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

__vmalloc_node_flags()

mm/vmalloc.c

```
static inline void *__vmalloc_node_flags(unsigned long size,
                                        int node, gfp_t flags)
{
        return __vmalloc_node(size, 1, flags, PAGE_KERNEL,
                                        node, __builtin_return_address
(0));
}
```

It allocates contiguous virtual memory from the requesting node for the kernel.

__vmalloc_node()

mm/vmalloc.c

```
/**
 *      __vmalloc_node  -  allocate virtually contiguous memory
 *      @size:          allocation size
 *      @align:         desired alignment
 *      @gfp_mask:      flags for the page level allocator
 *      @prot:          protection mask for the allocated pages
 *      @node:          node to use for allocation or NUMA_NO_NODE
 *      @caller:        caller's return address
 *
 *      Allocate enough pages to cover @size from the page level
 *      allocator with @gfp_mask flags.  Map them into contiguous
 *      kernel virtual space, using a pagetable protection of @prot.
 *
 *      Reclaim modifiers in @gfp_mask - __GFP_NORETRY, __GFP_RETRY_MAYFAIL
 *      and __GFP_NOFAIL are not supported
 *
 *      Any use of gfp flags outside of GFP_KERNEL should be consulted
 *      with mm people.
 *
 */
```

```
static void *__vmalloc_node(unsigned long size, unsigned long align,
                            gfp_t gfp_mask, pgprot_t prot,
                            int node, const void *caller)
{
        return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
                                gfp_mask, prot, 0, node, caller);
}
```

For the kernel, the @node allocates contiguous virtual memory, but the virtual address uses the empty space in the VMALLOC address space to map the virtual address.

__vmalloc_node_range()

mm/vmalloc.c

```
/**
 *      __vmalloc_node_range  -  allocate virtually contiguous memory
 *      @size:          allocation size
 *      @align:         desired alignment
 *      @start:         vm area range start
 *      @end:           vm area range end
 *      @gfp_mask:      flags for the page level allocator
 *      @prot:          protection mask for the allocated pages
 *      @vm_flags:      additional vm area flags (e.g. %VM_NO_GUARD)
 *      @node:          node to use for allocation or NUMA_NO_NODE
 *      @caller:        caller's return address
 *
 *      Allocate enough pages to cover @size from the page level
 *      allocator with @gfp_mask flags.  Map them into contiguous
 *      kernel virtual space, using a pagetable protection of @prot.
 */
```

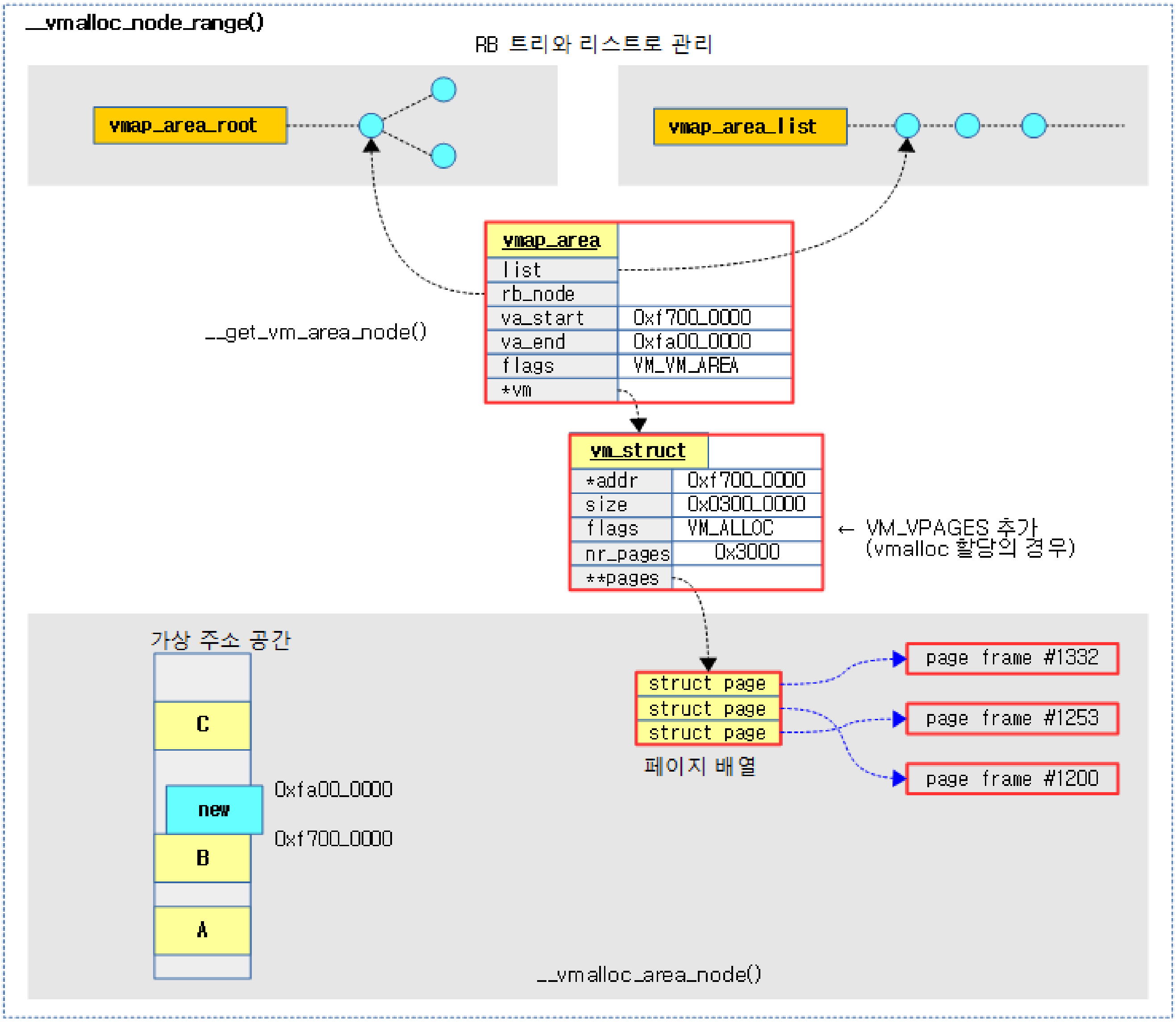
```
void *__vmalloc_node_range(unsigned long size, unsigned long align,
                        unsigned long start, unsigned long end, gfp_t gfp_mask,
03                          pgprot_t prot, unsigned long vm_flags, int node,
04                          const void *caller)
05  {
06          struct vm_struct *area;
07          void *addr;
08          unsigned long real_size = size;
09  
10          size = PAGE_ALIGN(size);
11          if (!size || (size >> PAGE_SHIFT) > totalram_pages)
12                  goto fail;
13  
14          area = __get_vm_area_node(size, align, VM_ALLOC | VM_UNINITIALIZED |
15                                  vm_flags, start, end, node, gfp_mask, caller);
16          if (!area)
17                  goto fail;
18  
19          addr = __vmalloc_area_node(area, gfp_mask, prot, node);
20          if (!addr)
21                  return NULL;
22  
23          /*
24           * In this function, newly allocated vm_struct has VM_UNINITIALIZED
25           * flag. It means that vm_struct is not fully initialized.
26           * Now, it is fully initialized, so remove this flag here.
27           */
28          clear_vm_uninitialized_flag(area);
29  
30          /*
31           * A ref_count = 2 is needed because vm_struct allocated in
32           * __get_vm_area_node() contains a reference to the virtual address of
33           * the vmalloc'ed block.
34           */
35          kmemleak_vmalloc(area, size, gfp_mask);
36  
37          return addr;
38  
39  fail:
40          warn_alloc_failed(gfp_mask, NULL,
41                            "vmalloc: allocation failure: %lu bytes\n", real_size);
42          return NULL;
43  }
```

Allocate contiguous virtual memory on the request node, but let the virtual address map the virtual address using the empty space within the specified range.

- In lines 10~12 of the code, the size is sorted by page and if the size is 0 or greater than the total memory, it returns null via the fail label.
- In lines 14~17 of the code, find a blank space in the request virtual address range where the request size can be placed, configure the vmap_area and vm_struct information to that virtual address, and return it.
  - See: Vmap (http://jake.dothome.co.kr/vmap) | Qc
- In code lines 19~21, the page descriptor table is allocated for the virtual address area that the vm_struct information is requested, and the order 0 pages are assigned to the number of requests to be connected and mapped to the page table.
- In line 28 of the code, clear the uninitialized flag for the VM to indicate that the VM has been initialized.

- Returns the virtual address space allocated in line 37 of code.

The following figure shows how a vmap_area and a vm_struct are allocated and constructed, and the required number of physical pages is assigned a page descriptor array, and that number of order 0 pages are allocated to connect and mapped.



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

__vmalloc_area_node()

mm/vmalloc.c

```
static void *__vmalloc_area_node(struct vm_struct *area, gfp_t gfp_mask,
                                 pgprot_t prot, int node)
{
        struct page **pages;
        unsigned int nr_pages, array_size, i;
        const gfp_t nested_gfp = (gfp_mask & GFP_RECLAIM_MASK) | __GFP_Z
ERO;
        const gfp_t alloc_mask = gfp_mask | __GFP_NOWARN;
        const gfp_t highmem_mask = (gfp_mask & (GFP_DMA | GFP_DMA32)) ?
                                        0 :
                                        __GFP_HIGHMEM;

12 	nr_pages = get_vm_area_size(area) >> PAGE_SHIFT;
13 	array_size = (nr_pages * sizeof(struct page *));
14 
15 	area->nr_pages = nr_pages;
16 	/* Please note that the recursion is strictly bounded. */
17 	if (array_size > PAGE_SIZE) {
18 		pages = __vmalloc_node(array_size, 1, nested_gfp|highmem_mask,
19 				PAGE_KERNEL, node, area->caller);
20 	} else {
21 		pages = kmalloc_node(array_size, nested_gfp, node);
22 	}
23 	area->pages = pages;
24 	if (!area->pages) {
25 		remove_vm_area(area->addr);
26 		kfree(area);
27 		return NULL;
28 	}
29 
30 	for (i = 0; i < area->nr_pages; i++) {
31 		struct page *page;
32 
33 		if (node == NUMA_NO_NODE)
34 			page = alloc_page(alloc_mask|highmem_mask);
35 		else
36 			page = alloc_pages_node(node, alloc_mask|highmem_mask, 0);
37 
38 		if (unlikely(!page)) {
39 			/* Successfully allocated i pages, free them in __vunmap() */
40 			area->nr_pages = i;
41 			goto fail;
42 		}
43 		area->pages[i] = page;
44 		if (gfpflags_allow_blocking(gfp_mask|highmem_mask))
45 			cond_resched();
46 	}
47 
48 	if (map_vm_area(area, prot, pages))
49 		goto fail;
50 	return area->addr;
51 
52 fail:
53 	warn_alloc(gfp_mask, NULL,
54 			  "vmalloc: allocation failure, allocated %ld of %ld bytes",
55 			  (area->nr_pages*PAGE_SIZE), area->size);
56 	vfree(area->addr);
57 	return NULL;
58 }
```

vm_struct is allocated an array of page descriptors for the virtual address area that the information is requesting, and a single page is allocated to the number of requests to be connected and mapped to a page table.

- In line 6 of code, add a __GFP_ZERO, leaving only the flag related to page recall.
- In line 7 of code, add a __GFP_NOWARN to the gfp_mask.
- If there is no request for DMA and DMA8 zones in code lines 10~32, prepare a highmem zone mask.
- In line 12 of the code, we find out how many pages the zone uses and find the size of the array that the page descriptors will use.
- In line 15 of the code, write down the number of pages that will be used for the VM.

- In lines 17~19 of code, if the array_size for page descriptor array allocation is more than 1 page, array_size amount of space is allocated in the zone containing the highmem of that node.
    - The vmalloc() function is nested and called while it is in progress.
    - Set a VM_VPAGES to the area's flag information to indicate that there is an array of page descriptors assigned to it.
- In lines 20~22 of the code, if the array_size is less than 1 page array_size, use the kmalloc_node() function to assign a slab object to construct an array of page descriptors.
- In lines 23~28 of the code, point to the array of page descriptors used by area. (VM only mem_map)
- In line 30~46 of the code, loop around the number of pages, get 1 single page from the page allocator using the buddy system, and connect each assigned page to the page descriptor array.
- Perform the mapping of the area in lines 48~50 of the code and return its virtual address. If it fails, move it to the fail label.

clear_vm_uninitialized_flag()

mm/vmalloc.c

```
static void clear_vm_uninitialized_flag(struct vm_struct *vm)
{
        /*
         * Before removing VM_UNINITIALIZED,
         * we should make sure that vm has proper values.
         * Pair with smp_rmb() in show_numa_info().
         */
        smp_wmb();
        vm->flags &= ~VM_UNINITIALIZED;
}
```

Remove the VM_UNINITIALIZED flag to indicate that the VM setup is complete.

deallocate vmalloc

vfree()

mm/vmalloc.c

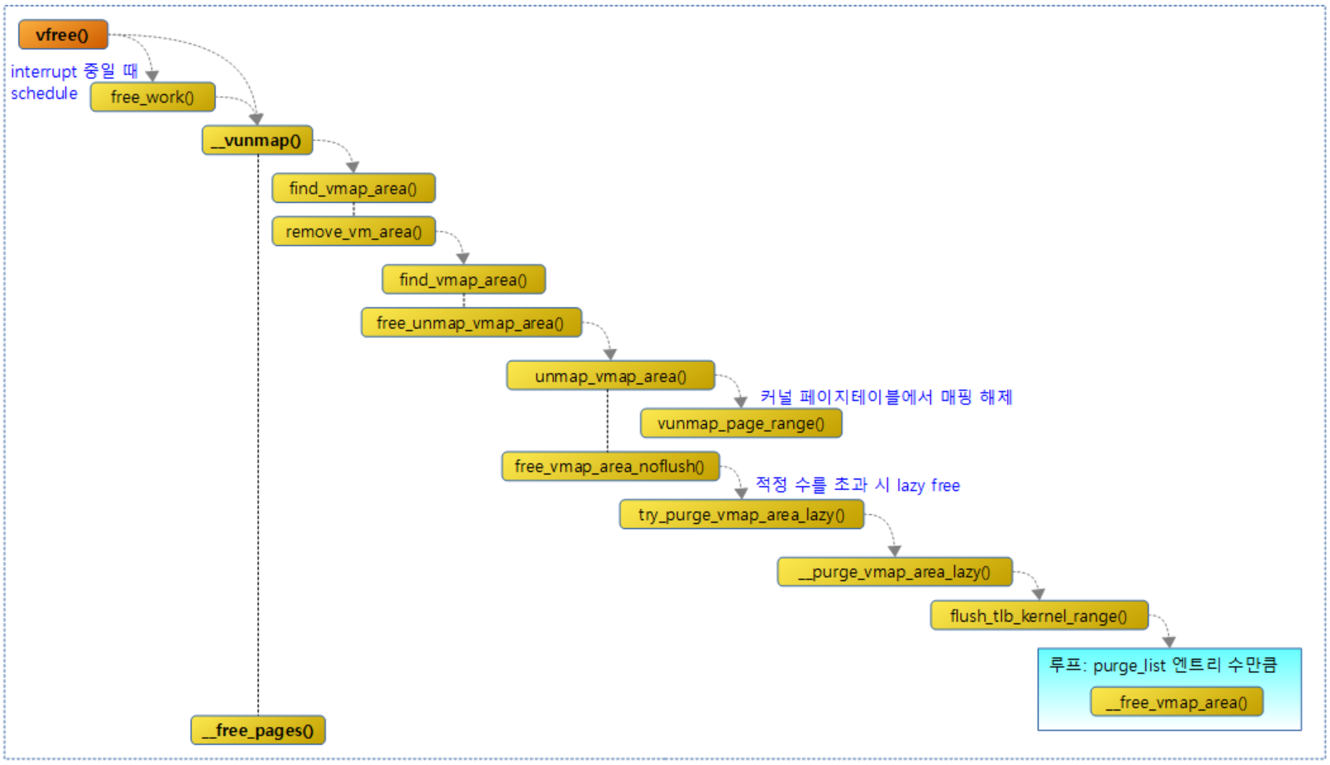
```
/**
 *      vfree  -  release memory allocated by vmalloc()
 *      @addr:          memory base address
 *
 *      Free the virtually continuous memory area starting at @addr, as
 *      obtained from vmalloc(), vmalloc_32() or __vmalloc(). If @addr i
s
 *      NULL, no operation is performed.
 *
 *      Must not be called in NMI context (strictly speaking, only if we
don't
 *      have CONFIG_ARCH_HAVE_NMI_SAFE_CMPXCHG, but making the calling
 *      conventions for vfree() arch-depenedent would be a really bad id
ea)
 *
 *      May sleep if called *not* from interrupt context.
 *
 *      NOTE: assumes that the object at @addr has a size >= sizeof(llist_node)
 */
```

```
void vfree(const void *addr)
{
        BUG_ON(in_nmi());

        kmemleak_free(addr);

        might_sleep_if(!in_interrupt());

        if (!addr)
                return;
        if (unlikely(in_interrupt()))
                __vfree_deferred(addr);
        else
                __vunmap(addr, 1);
}
EXPORT_SYMBOL(vfree);
```

vmalloc() frees up contiguous virtual address memory allocated and mapped.

- If it is called via an interrupt handler, it is not possible to release the mapped contiguous virtual address memory right away, so it is added to the per-cpu vfree_deferred for deferred processing, and then the free_work() function registered in the workqueue is scheduled.

The following illustration shows the flow of the vfree() function and its associated functions.



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

free_work()

mm/vmalloc.c

```
static void free_work(struct work_struct *w)
{
        struct vfree_deferred *p = container_of(w, struct vfree_deferred, wq);
        struct llist_node *t, *llnode;

        llist_for_each_safe(llnode, t, llist_del_all(&p->list))
                __vunmap((void *)llnode, 1);
}
```

Free up all VMs (memory mapped to contiguous virtual addresses) in the per-CPU vfree_deferred list.

consultation

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

- 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
- Red/Black Tree Visualization (https://www.cs.usfca.edu/~galles/visualization/RedBlack.html)

4 thoughts to “Vmalloc”

IPARAN (HTTPS://WWW.BHRAL.COM/)
2022-01-15 21:56 (http://jake.dothome.co.kr/vmalloc/#comment-306240)

Hello! Moon Young-il, this is the 16th Iparan. Thank you always~

__vmalloc_area_node()

/* Please note that the recursion is strictly bounded. */
if (array_size > PAGE_SIZE) {
pages = __vmalloc_node(array_size, 1, nested_gfp|highmem_mask,
PAGE_KERNEL, node, area->caller);
}

In the comments, recursion doesn't happen much anyway. It says.
If the size you are requesting dynamic is large, you can get it from Buddy, but I don't know why you are doing it with vmalloc, so I am asking!
When you do dynamic allocation, do you want to use a limited number of physically contiguous physical pages?

RESPONSE (/VMALLOC/?REPLYTOCOM=306240#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2022-01-18 11:01 (http://jake.dothome.co.kr/vmalloc/#comment-306244)

Hello? Mr. Iparan!

Essentially, pages assigned using the vmalloc method are used to minimize fragmentation in the buddy system.

vmalloc allows virtual pages to be continuous, but physical pages don't have to be continuous.
On the other hand, the pages allocated by the Buddy system are very expensive resources that require both virtual and physical pages to be contiguous.
This takes more time to allocate than using the buddy system directly, but vmalloc is used to prevent fragmentation.

I appreciate it.

RESPONSE (/VMALLOC/?REPLYTOCOM=306244#RESPOND)

IPARAN (HTTP://WWW.BHRAL.COM)
2022-01-18 19:12 (http://jake.dothome.co.kr/vmalloc/#comment-306245)

Thank you for your reply~
If I modify the function with buddy allocation and then the physical/memory allocation happens in high order, if
you are unlucky, you may have to squeeze out the expensive resource of contiguous memory (page recall, memory compaction), etc.
If you're unlucky, why not use discrete memory?

RESPONSE (/VMALLOC/?REPLYTOCOM=306245#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2022-01-19 09:23 (http://jake.dothome.co.kr/vmalloc/#comment-306246)

네. 그렇습니다.

vmalloc 자체가 버디 시스템의 high order를 사용하지 않도록 한 것이니깐요.
해당 코드를 바꾸면 운영 중 OOM이 발생할 확률이 더 커집니다. ^^

감사합니다.

응답 (/VMALLOC/?REPLYTOCOM=306246#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

< vmalloc_init() (http://jake.dothome.co.kr/vmalloc_init/)

Vmap > (http://jake.dothome.co.kr/vmap/)

문c 블로그 (2015 ~ 2024)