vmalloc

vmalloc이 하는 일은 잘 알다시피 virtually contiguous memory 영역을 할당하는 함수이다.
이 함수는 __vmalloc_node를 호출하게 된다.

__vmalloc_node

```
static void *__vmalloc_node(unsigned long size, gfp_t gfp_mask, pgprot_t prot,
                            int node, void *caller)
```

이 함수는 __get_vm_area_node를 호출하여 필요한 작업들을 하고 vm_struct 을 생성하게 된다.
그런 후, __vmalloc_area_node를 호출하여 vm_struct에 필요한 page들을 채우고 반환하게 된다.

__get_vm_area_node

```
static struct vm_struct *__get_vm_area_node(unsigned long size,
        unsigned long flags, unsigned long start, unsigned long end,
        int node, gfp_t gfp_mask, void *caller)
```

이 함수가 실제 필요한 모든 작업을 하는 함수이다.
먼저, slab 으로부터 vm_struct을 할당받는다.
사용자가 요청한 page 크기에 PAGE_SIZE를 추가하여 Guard page의 공간을 만들게 된다.
alloc_vmap_area를 호출하여 vmap_area를 할당한다. 이때 할당된 vmap_area는 vmap_area_root의 rbtree와 vmap_area_list의 연결 리스트에 연결된다.
vm_struct의 여러 필드들을 초기화 한다. 이때 중요한 것은 alloc_vmap_area를 통해 할당받은 vmap_area의 private 필드를 vm_struct로 지정한다는 것과 va->flags에 VM_VM_AREA를 지정한다는 것이다.
마지막으로 vmlist를 보호하는 rwlock인 vmlist_lock을 write_lock으로 잡아 보호한 후, vmlist를 선형으로 검색하여 연결
write_unlock

alloc_vmap_area

```
static struct vmap_area *alloc_vmap_area(unsigned long size,
                unsigned long align,
                unsigned long vstart, unsigned long vend,
                int node, gfp_t gfp_mask)
```

slab 으로부터 vmap_area를 할당받는다.
vmap_area_lock의 spin_lock을 잡고 vmap_area_root를 순회하여 size에 해당하는 빈 공간을 찾아내어 vmap_area의 변수들에 지정하고 __insert_vmap_area를 호출하여 vmap_area_root에 연결하고 spin_unlock으로 vmap_area_lock을 해지한다.
먼저 vmap_area_root를 순회하여 사용자가 원하는 가상 주소 공간의 크기만큼 빈 공간이 있는지 확인한다. 처음 한번 순회한 후, 빈 공간이 없다면 purge_vmap_area_lazy 함수를 호출하여 이전에 사용된 후, 해지된 vmap_area들을 flush하고 다시 한번 주소 공간을 찾게 된다. purge_vmap_area_lazy 함수는 아래에서 더욱 자세히 살펴보기로 한다.

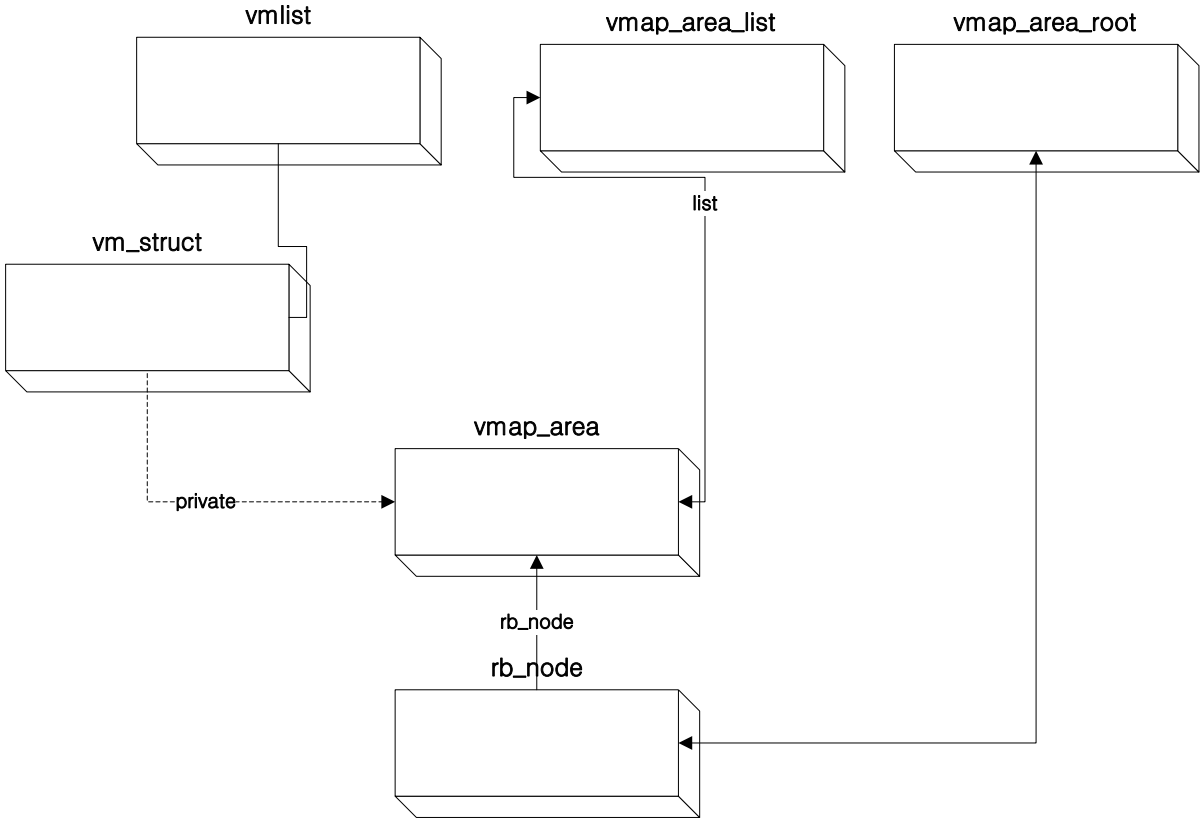
__insert_vmap_area

```
static void __insert_vmap_area(struct vmap_area *va)
```

처음에는 vmap_area_root.rb_node가 비어있기 때문에 바로 집어넣게 된다.
rb_link_node를 호출하여 넣게 되고, rb_insert_color를 호출하여 rbtree의 균형을 잡는다.
그런 후, rb_node의 rb_prev가 없기 때문에 list_add_rcu를 호출하여 vmap_area_list에 va를 연결한다. 즉, RCU를 vmap_area_list에 연결하기 위해서 사용한다.

한번 더 살펴보자. 이전까지는 rbtree의 root가 비어있을 경우 였고, 이번에는 하나가 추가되었으니 rbtree를 검색해보자. 다시 alloc_vmap_area가 호출되었다고 가정하자. 이번에는 vmap_area_lock의 spin_lock을 잡고 rbtree를 순회한다. size가 32였다고 가정하고 VMALLOC_START가 0xc0000000이라고 가정하자. 첫번째 순회에서는 tmp->va_end는 0xc0000020이다. 그리고 vmp->va_start는 0xc0000000이다. 그러므로 first = tmp가 실행된다.
그리고 나서 n->rb_left는 NULL이므로 루프를 빠져나간다.
first는 NULL이 아니므로 그 다음 if문으로 이동한다. first->va_end는 addr보다 크기 때문에 다음 while 문으로 이동하게 된다. addr + size는 당연히 first->va_start보다 크고 addr+ size는 vend와 같기 때문에 while의 body로 들어오게 되며, first->va_end + PAGE_SIZE를 align한 주소 addr을 얻게 된다. 즉, 0xc0000020 + 0x1000을 32에 align하게 되면 0xc0001020이 된다. 그런 후 first의 rb_next이 없기 때문에 found로 이동하여 va를 rbtree에 넣게 된다.

alloc_vmap_area 함수의 호출이 종료되면 다음과 같이 자료구조들이 연결된다.

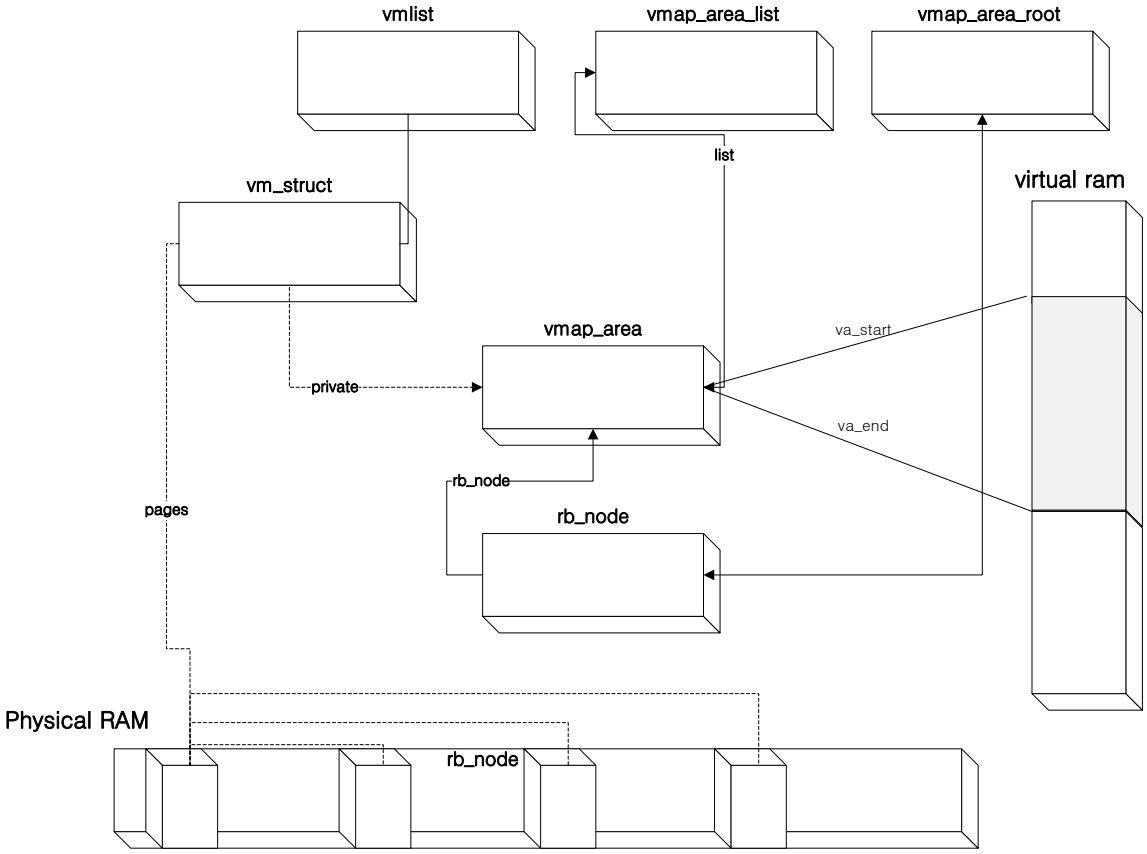


__get_vm_area_node가 위와 같은 자료구조를 생성하고 나면 __vmalloc_area_node가 호출되어 가상 주소 영역과 관련된 page들을 생성한 후, vm_struct에 연결하고 페이지들을 매핑하게 된다.

__vmalloc_area_node

```
static void *__vmalloc_area_node(struct vm_struct *area, gfp_t gfp_mask,
                                 pgprot_t prot, int node, void *caller)
```

먼저, 이 가상 주소 공간에 속하게 될 물리 page의 수가 얼마인지 파악한다. 그리고 이 page들을 위한 struct page 구조체가 얼마나 필요하게 될지 파악하여 그 크기가 한 page 이상 된다면 __vmalloc_node를 호출하여 page 디스크립터들의 배열을 생성하고 그렇지 않고 한 page내에 들어가는 크기라면 kmalloc_node를 호출하게 된다. 이때 kmalloc_node를 __GFP_ZERO를 통해서 0으로 초기화된 영역을 생성하여 vm_struct와 연결하게 된다.



위의 그림에서 보는 것과 같이 먼저 page descriptor들을 저장하기 위한 page를 할당한 후(정확하게는 one page를 할당하는 것이 아니라 slab을 사용하게 된다.), vmalloc의 사용자에 의해 사용될 page들을 할당하여 연결하게 된다. 이때 page 디스크립터들을 위한 페이지 할당은 성공했지만 user 사용을 위한 page 할당을 충분히 만족시키지 못하고 실패하였을 경우 vfree를 호출하여 그 동안 생성하였던 모든 자료구조들을 free한다. vfree는 다음에서 살펴보기로 한다.

map_vm_area

다음으로 map_vm_area를 호출하여 연결된 page들을 커널 페이지 테이블(init_mm->pgd)에 매핑하게 된다. 이때 주의해야 할 것이 있다. 많은 사람들이 착각하고 있는 것이 있다. 커널영역에서는 fault가 일어나지 않는다고 착각한다. vmalloc으로 할당한 영역도 이와 같이 초기에 페이지 테이블에 모두 매핑시키기 때문에 page fault가 발생하지 않는다고 생각한다. 하지만 page들이 매핑된 것은 init_mm->pgd, 즉 master page table이지 시스템에 수행중인 응용들의 kernel page table이 아니라는 것이다. 그러므로 커널이 process context에서 실행 도중 vmalloc으로 할당된 가상주소 공간을 접근하게 되면, 최초 한번은 demand paging이 일어날 수 밖에 없다.

vfree

vfree는 __vunmap을 호출한다.
__vunmap은 remove_vm_area를 호출하여 vm_area와 관련된 자료구조들을 정리한 후, 최종적으로 __free_page를 호출하여 가상 주소와 연결되어 있던 page과 page descriptor들을 저장하기 위한 slab object을 kfree를 호출하여 해지하고, 마지막으로 vm_struct을 해지한다.

remove_vm_area

```
struct vm_struct *remove_vm_area(const void *addr)
```

먼저, find_vmap_area를 호출하여 vmap_area 구조체를 찾는다. 이 함수는 위 그림에서 본 것과 같이 vmap_area_root의 rbtree에서 vmap_area를 찾게 된다. 이를 free_unmap_vmap_area를 호출하여 lazy free를 하게 된다. 그러므로 이 함수를 좀더 자세히 살펴보자.

```
static void free_unmap_vmap_area(struct vmap_area *va)
```

free_unmap_vmap_area_noflush

이 함수는 free_unmap_vmap_area_noflush를 호출하게 된다.
먼저, va->flags에 VM_LAZY_FREE를 명기한 후, vmap_lazy_nr에 vmap이 포함하는 가상 주소 크기에 해당하는 page 수만큼 더하여, vmap_lazy_nr이 lazy_max_pages보다 크다면 try_purge_vmap_area_lazy함수를 호출하여 flush하게 된다. VM_LAZY_FREE가 명기된 vmap_area들의 수가 lazy_max_pages보다 크게 되는 한꺼번에 free되며 flush될 것이다. 이와 같은 경우는 잠시 후 다시 살펴보자. 이번에는 크지 않다는 것을 가정하고 넘어가자.
다시 remove_vm_area로 돌아와서, vm_list에서 해당 vm_struct을 제거한다.

try_purge_vmap_area_lazy

이제 try_purge_vmap_area_lazy 함수를 살펴보기로 하자. 이 함수는 __purge_vmap_area_lazy 함수를 호출하게 된다. 이 함수가 실제 역할을 하게 된다.

```
static void __purge_vmap_area_lazy(unsigned long *start, unsigned long *end,
                                   int sync, int force_flush)
```

이 함수는 vmap_area_list를 순회하며, va->flags가 VM_LAZY_FREE인 vmap_area들에 대해

unmap_vmap_area를 호출하여 vmap_area가 cover하는 가상 주소들과 mapping되어 있는 kernel page table(init_mm)들을 모두 clear한다. 그런 후, va의 purge_list를 이용해서 valist에 va를 연결하고 va->flags에 VM_LAZY_FREE는 삭제하고, VM_LAZY_FREEING을 넣는다. 이런 식으로 vmap_area_list의 모든 va에 대해 위의 작업을 반복하고 flush_tlb_kernel_range를 호출하여 모든 프로세서들의 TLB를 flush하게 된다. 마지막으로 valist에 모아진 vmap_area들을 __free_vmap_area 함수를 통해서 해지하게 된다. 이때 __free_vmap_area는 vmap_area_list, vmap_area_root의 rbtree에서 va들의 연결을 제거하고, 마지막으로 call_rcu를 통해서 rcu_free_va함수가 vmap_area들을 free한다.

지금까지는 vmalloc의 2.6.28 이전의 interface에 대해 살펴보았다. 지금부터는 새로운 API인 vm_map_ram과 vm_unmap_ram에 대해 살펴보기로 하자. 그에 앞서 vmalloc_init 함수를 보자.

vmalloc_init 함수는 per_cpu 변수인 vmap_block_queue를 초기화한다.

vm_map_ram

```
void *vm_map_ram(struct page **pages, unsigned int count, int node, pgprot_t prot)
```

할당하려는 page의 수가 VMAP_MAX_ALLOC(32bit:32)보다 작거나 같다면 vb_alloc을 호출하게 된다. 먼저 vb_alloc에 대해 살펴보기로 하자.

vb_alloc

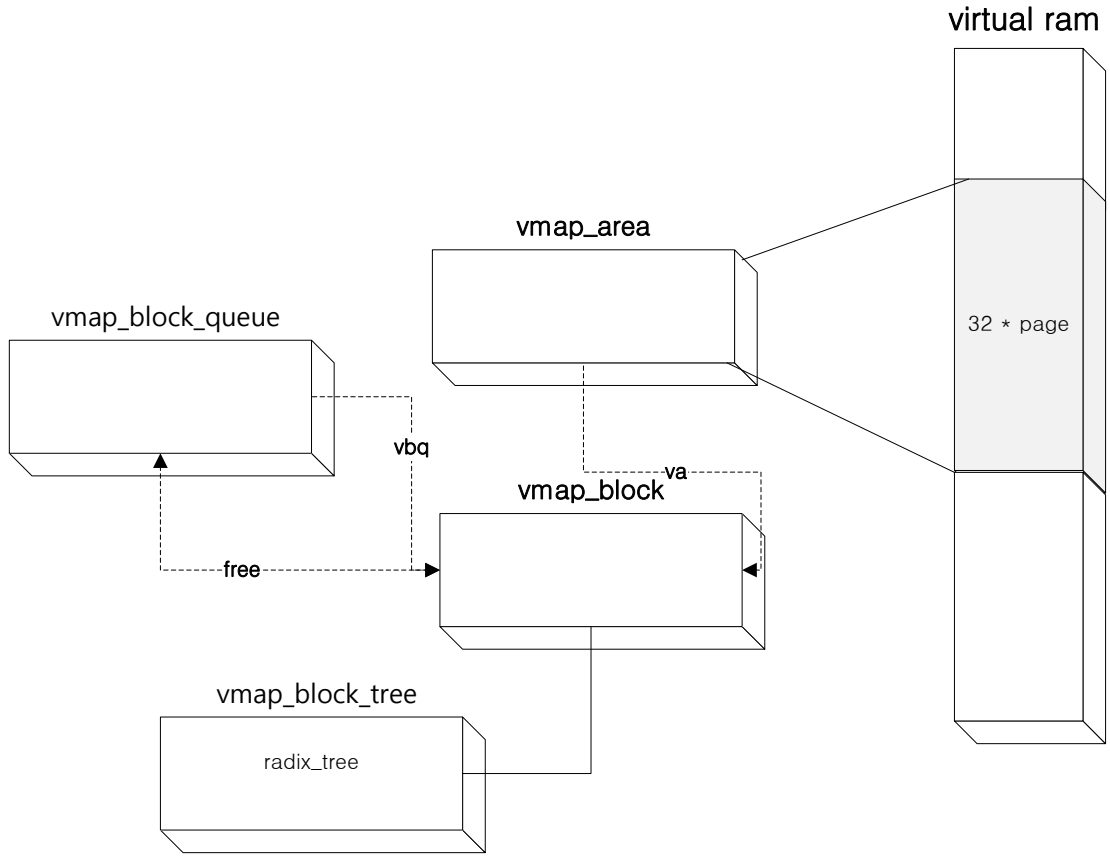
```
static void *vb_alloc(unsigned long size, gfp_t gfp_mask)
```

per_cpu 변수인 vmap_block_queue를 순회하는데 처음에는 freelist가 비어 있기 때문에 new_vmap_block을 호출하여 vmap_block을 할당받아 vmap_block_queue의 freelist에 연결한 후 다시 시도하게 된다.

new_vmap_block

```
static struct vmap_block *new_vmap_block(gfp_t gfp_mask)
```

이 함수는 먼저 vmap_block을 slab으로부터 할당받은 후, alloc_vmap_area를 호출하여 32 page 짜리 크기의 가상 주소 공간을 관리하는 vmap_area 구조체를 할당받아, vmap_block과 vmap_area를 연결한다. vmap_block의 필드들을 초기화 한 후, vmap_block을 vmap_block_tree의 radix tree에 연결한다. 그런 후, vmap_block을 per_cpu 변수인 vmap_block_queue와 연결하고 vmap_block_queue의 free_list에 연결시킨다.



다시 vb_alloc 함수로 돌아가 위의 함수에서 vmap_block_queue의 freelist에 vmap_block을 연결시켰음으로 다시 한번 search한다. 현재 cpu의 vmap_block_queue의 free_list의 vmap_block들을 순회하된다. 이때 vb_alloc에 요청한 크기만큼의 가상 주소 공간을 찾게 된다. 이 역할을 하는 함수가 bitmap_find_free_region이다. 이 함수를 통해 찾게 된 시작 주소를 반환한다.

다시 vm_map_ram 함수로 돌아오자. 위에서 vb_alloc 함수는 vmap_block을 할당하여 vmap_block_queue의 freelist 에 연결시킨 후, 사용자가 요청한 크기 만큼의 가상주소공간을 확보하여 시작주소를 반환하였다. vm_map_ram 함수는 마지막으로 할당받은 주소를 vmap_page_range 함수를 통하여 인수로 받은 page들을 가상 주소 공간에 연결하게 된다.

이번에는 vm_map_ram 함수로 할당받으려는 크기가 VMAP_MAX_ALLOC보다 큰 경우를 살펴보자. 이때는 vb_alloc 함수가 아닌 앞서 살펴보았던 alloc_vmap_area 함수를 직접 호출하여 vmap_area를 할당받게 된다. 결국 이 두 함수의 차이는 vb_alloc은 각 CPU에 캐시된(32bit : 32 64bit : 64개의 page) vmap_area의 가상 주소 공간을 나누어 쓴다는 것이다. 그러므로 할당 면에 있어서나 allocation performance에 있어서나 lock 측면에 있어서 훨씬 이득이다.

vm_unmap_ram

이번에는 vm_unmap_ram 함수를 살펴보자.

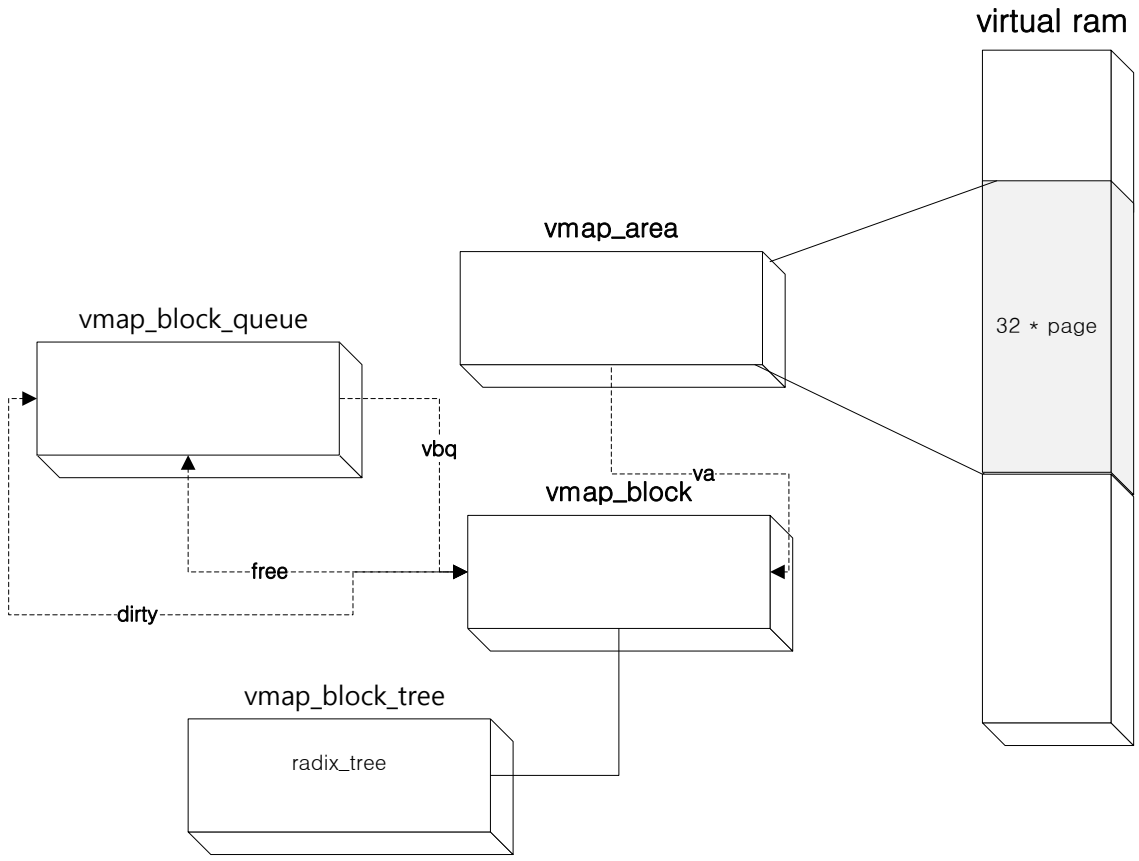
void vm_unmap_ram(const void *mem, unsigned int count)

이 함수도 vm_map_ram 함수와 마찬가지로 해지되는 memory의 크기가 VMAP_MAC_ALLOC보다 작거나 같다면 vb_free를 호출한다. 그렇지 않은 경우는 앞서 살펴 보았던 free_unmap_vmap_area_addr을 호출하여 해지하게 된다. 그러므로 여기서는 vb_free 함수만을 살펴보기로 한다.

vb_free

static void vb_free(const void *addr, unsigned long size)

vmap_block_tree의 radixtree로부터 vmap_block을 찾아낸다. vmap_block을 vmap_block_queue의 dirty list에 연결한다.



dirty_list가 꽉 차게 되면 그때 free_vmap_block을 호출하여 radix tree에서 vmap_block을 제거하고 vmap_block에 연결된 vmap_area를 free_unmap_vmap_area_noflush를 호출하여 해지하고 rcu_free_vb를 호출하여 vmap_block을 free 하게 된다.