Chapter 4

메모리 관리

18 January, 2021

Minguk Choi

koreachoi96@gmail.com

Contents

- Physical Memory Data Structure
    - UMA/NUMA
    - Node, Zone, Page

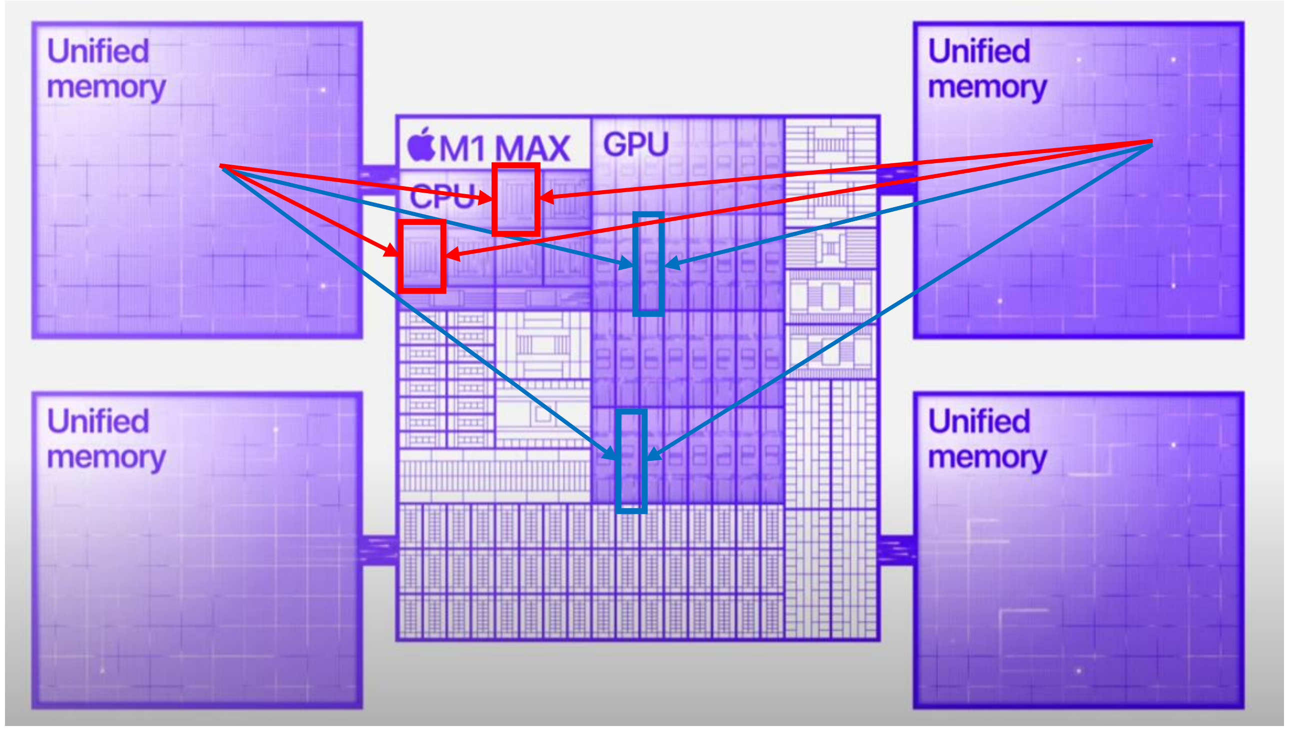
- Buddy Allocator
    - alloc_pages / free_pages()
    - vmalloc() / vfree()
    - practice1: alloc_pages() vs vmalloc()

- Slab Allocator
    - kmalloc() / kfree()
    - practice2: kmem_cache_alloc() vs kmalloc()

- Kernel Stack
    - practice3: kernel stack overflow

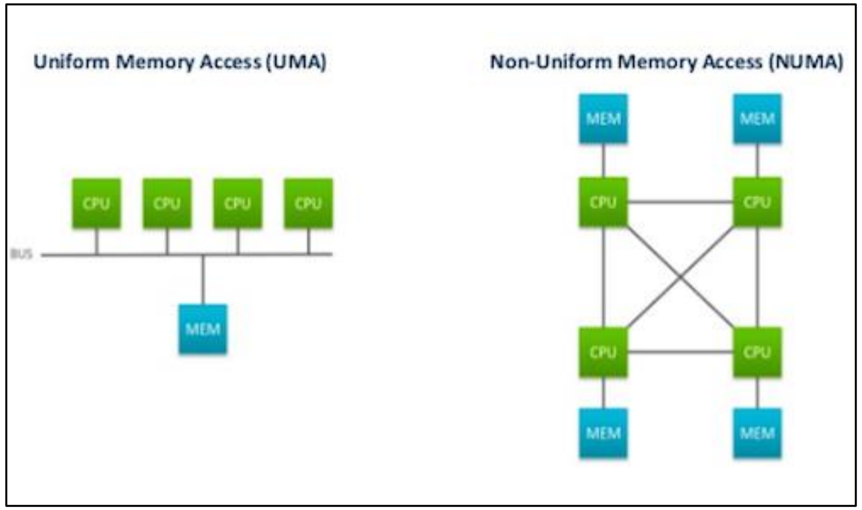
Physical Memory Data Structure

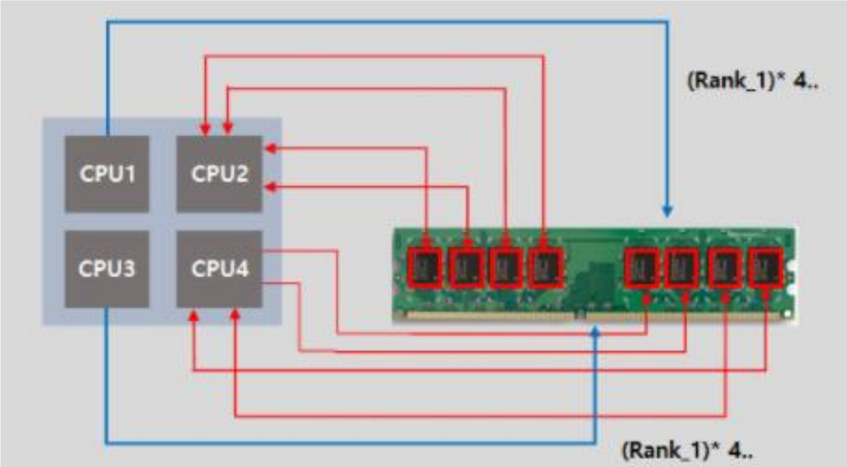
- Apple M1 Max SoC



Physical Memory Data Structure

- SMP/NUMA



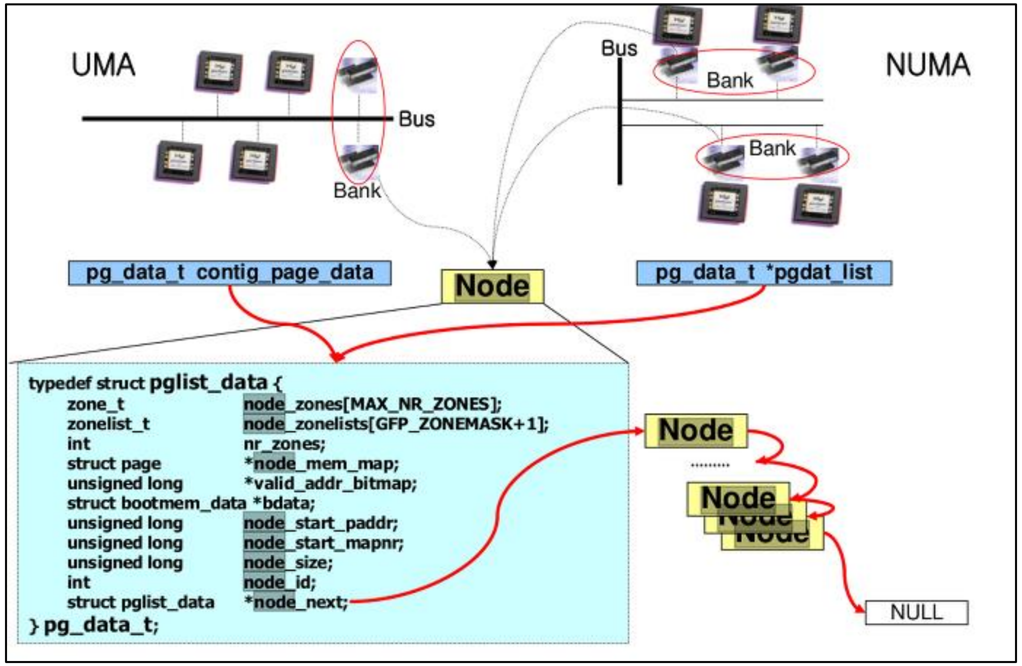


- SMP(UMA)
  - ✓ Symmetric Multiprocessing
  - ✓ 모든 CPU와 메모리가 하나의 입출력 버스 등을 공유
  - ✓ 병목현상 발생

- NUMA
  - ✓ Non-Uniform Memory Access
  - ✓ CPU를 그룹으로 나누고, 각 그룹에 별도에 지역 메모리 할당
  - ✓ CPU가 어떤 메모리를 접근하느냐에 따라 성능차이 발생

Physical Memory Data Structure

- Node



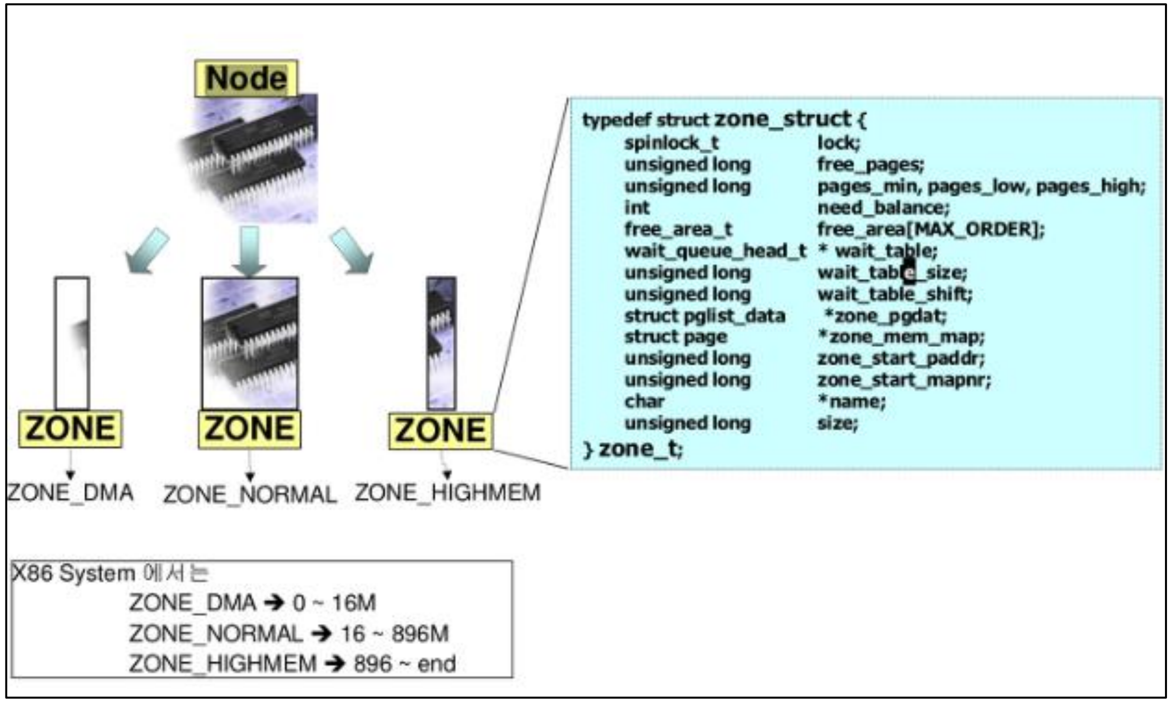
```
typedef struct pglist_data {
        zone_t                  node_zones[MAX_NR_ZONES];
        zonelist_t              node_zonelists[GFP_ZONEMASK+1];
        int                     nr_zones;
        struct page             *node_mem_map;
        unsigned long           *valid_addr_bitmap;
        struct bootmem_data *bdata;
        unsigned long           node_start_paddr;
        unsigned long           node_start_mapnr;
        unsigned long           node_size;
        int                     node_id;
        struct pglist_data      *node_next;
} pg_data_t;
```

- Bank
  - ✓ 리눅스에서 접근 속도가 같은 메모리의 집합
  - ✓ UMA: single Bank
  - ✓ NUMA: non-single Banks

- Node
  - ✓ 리눅스에서 Bank를 표현하는 자료구조

Physical Memory Data Structure

- Zone



```
typedef struct zone_struct {
      spinlock_t              lock;
      unsigned long           free_pages;
      unsigned long           pages_min, pages_low, pages_high;
      int                     need_balance;
      free_area_t             free_area[MAX_ORDER];
      wait_queue_head_t  * wait_table;
      unsigned long           wait_table_size;
      unsigned long           wait_table_shift;
      struct pglist_data       *zone_pgdat;
      struct page             *zone_mem_map;
      unsigned long           zone_start_paddr;
      unsigned long           zone_start_mapnr;
      char                    *name;
      unsigned long           size;
} zone_t;
```

X86 System 에서는
ZONE_DMA ➔ 0 ~ 16M
ZONE_NORMAL ➔ 16 ~ 896M
ZONE_HIGHMEM ➔ 896 ~ end

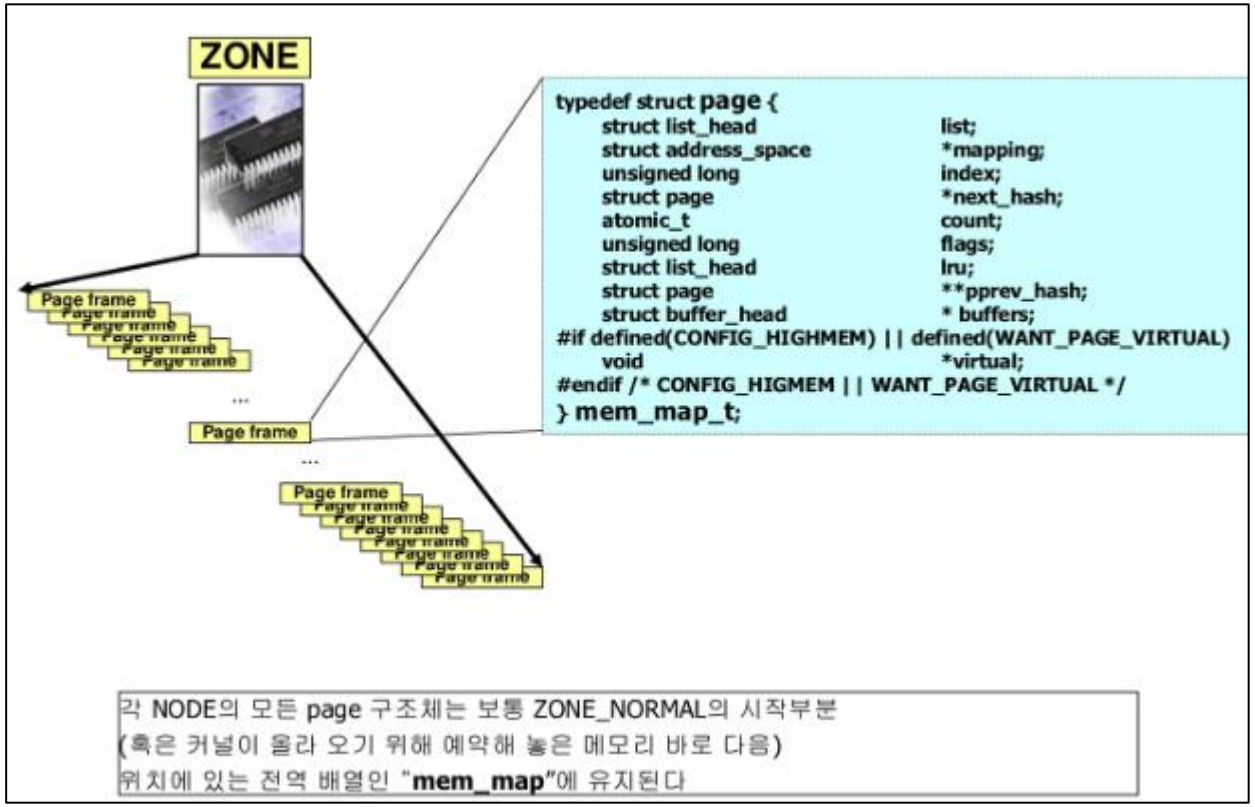
- 커널이 모든 메모리를 동일하게 취급?
  - ✓ 일부 하드웨어의 한계로 불가
    - 특정 메모리 주소로만 DMA 수행가능
    - 일부 메모리는 커널 주소 공간에 상주 불가

- ZONE
  - ✓ ZONE_DMA: DMA 수행가능
  - ✓ ZONE_NORMAL
  - ✓ ZONE_HIGHMEM: 커널 주소공간 상주X
    페이지 동적으로 연결

- 32bit-x86
  - ✓ ZONE_DMA: 0MB ~ 16MB
  - ✓ ZONE_NORMAL: 16MB ~ 896MB
  - ✓ ZONE_HIGHMEM: 896MB ~

Physical Memory Data Structure

- Page



```
typedef struct page {
      struct list_head                list;
      struct address_space            *mapping;
      unsigned long                   index;
      struct page                     *next_hash;
      atomic_t                        count;
      unsigned long                   flags;
      struct list_head                lru;
      struct page                     **pprev_hash;
      struct buffer_head              * buffers;
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
      void                            *virtual;
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

각 NODE의 모든 page 구조체는 보통 ZONE_NORMAL의 시작부분
(혹은 커널이 올라 오기 위해 예약해 놓은 메모리 바로 다음)
위치에 있는 전역 배열인 "**mem_map**"에 유지된다

- 물리적 메모리 최소 관리단위
  - ✓ 커널: Page
  - ✓ Why? MMU(Memory Management Unit)
    - 페이지 단위로 가상 메모리 주소를 물리적 메모리 주소로 변환

- Page Size
  - ✓ 32bit: 4KB
  - ✓ 64bit: 8KB

- Struct page
  - ✓ 가상 페이지가 아닌, 물리적 페이지 표현
    - Struct page의 내용은 일시적 (ex. swap)

Physical Memory Data Structure

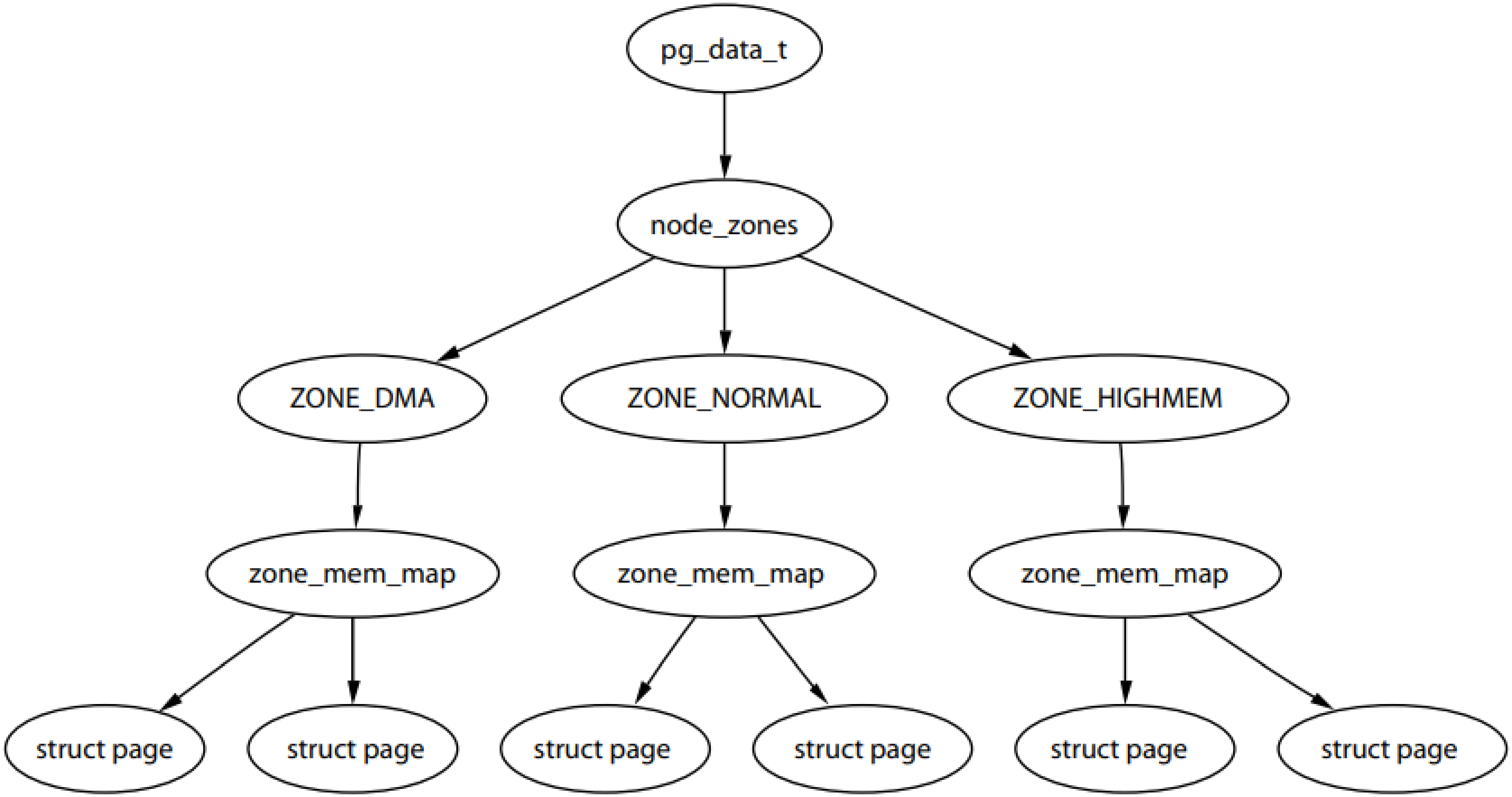
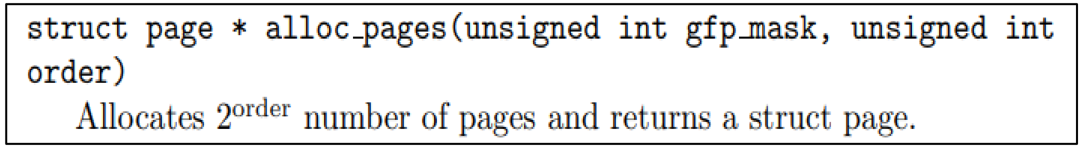


Figure 2.1. Relationship Between Nodes, Zones and Pages

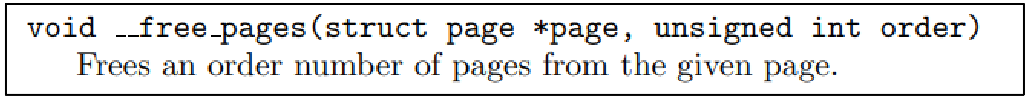
Buddy Allocator

- alloc_pages / free_pages()
  - contiguous VA, contiguous PA



```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int
order)
```

Allocates 2^{order} number of pages and returns a struct page.



```
void __free_pages(struct page *page, unsigned int order)
```

Frees an order number of pages from the given page.

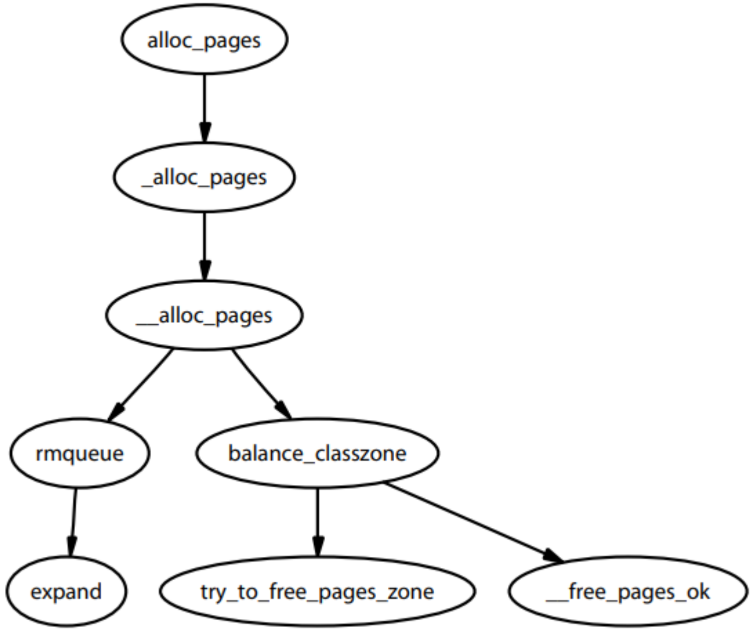


Figure 6.2. Call Graph: `alloc_pages()`

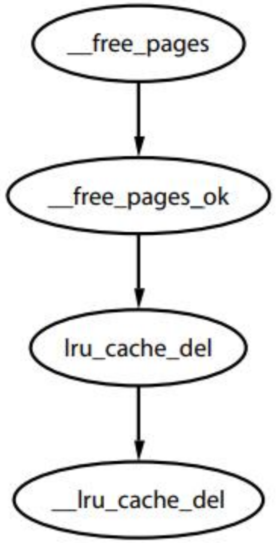
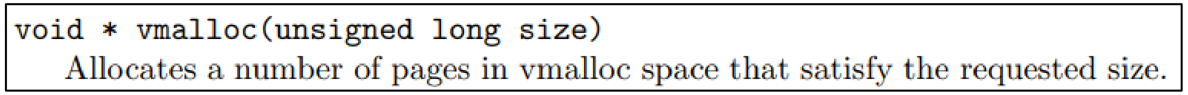


Figure 6.4. Call Graph: `__free_pages()`

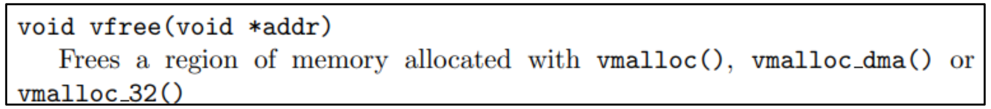
Buddy Allocator

- vmalloc() / vfree()
  - contiguous VA, but do not guarantee contiguous PA



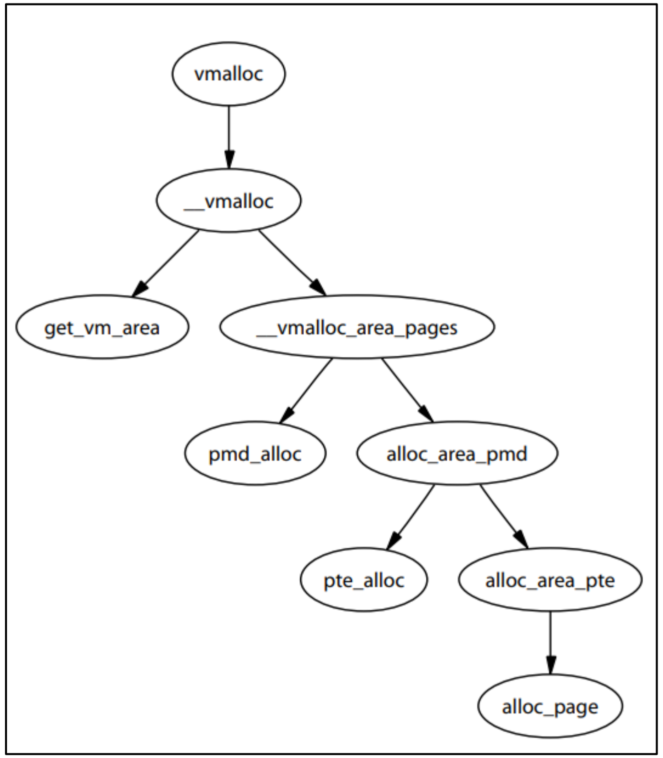
`void * vmalloc(unsigned long size)`

Allocates a number of pages in vmalloc space that satisfy the requested size.



`void vfree(void *addr)`

Frees a region of memory allocated with `vmalloc()`, `vmalloc_dma()` or `vmalloc_32()`



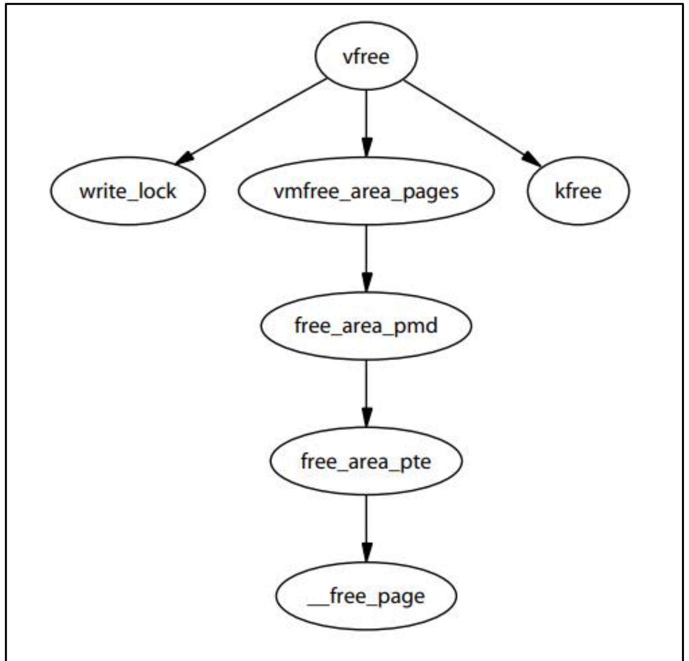


Figure 7.4. Call Graph: `vfree()`

Buddy Allocator

- vmalloc()

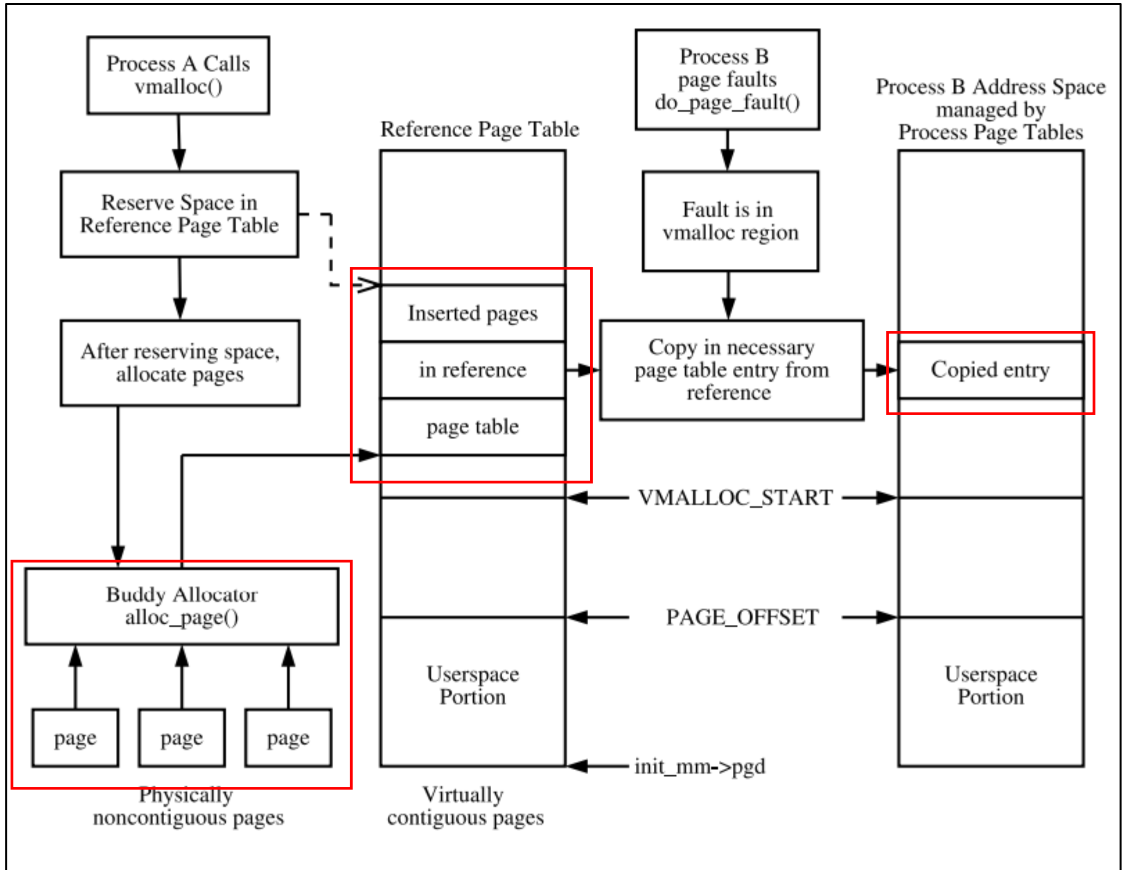

Process A Calls vmalloc()
Reserve Space in Reference Page Table
After reserving space, allocate pages
Buddy Allocator alloc_page()
page
page
page
Physically noncontiguous pages
Reference Page Table
Inserted pages
in reference
page table
Userspace Portion
Virtually contiguous pages
Process B page faults do_page_fault()
Fault is in vmalloc region
Copy in necessary page table entry from reference
VMALLOC_START
PAGE_OFFSET
init_mm->pgd
Process B Address Space managed by Process Page Tables
Copied entry
Userspace Portion

Figure 7.3. Relationship Between `vmalloc()`, `alloc_page()` and Page Faulting

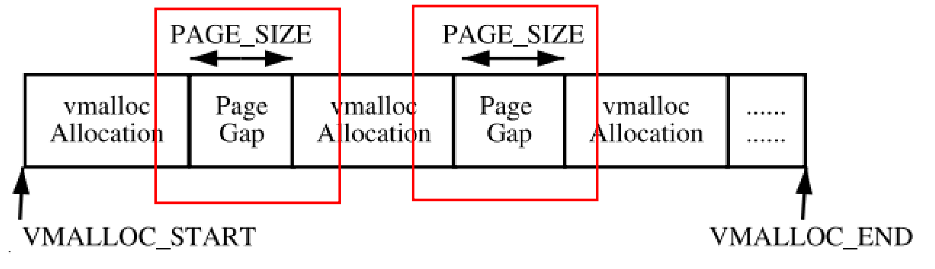
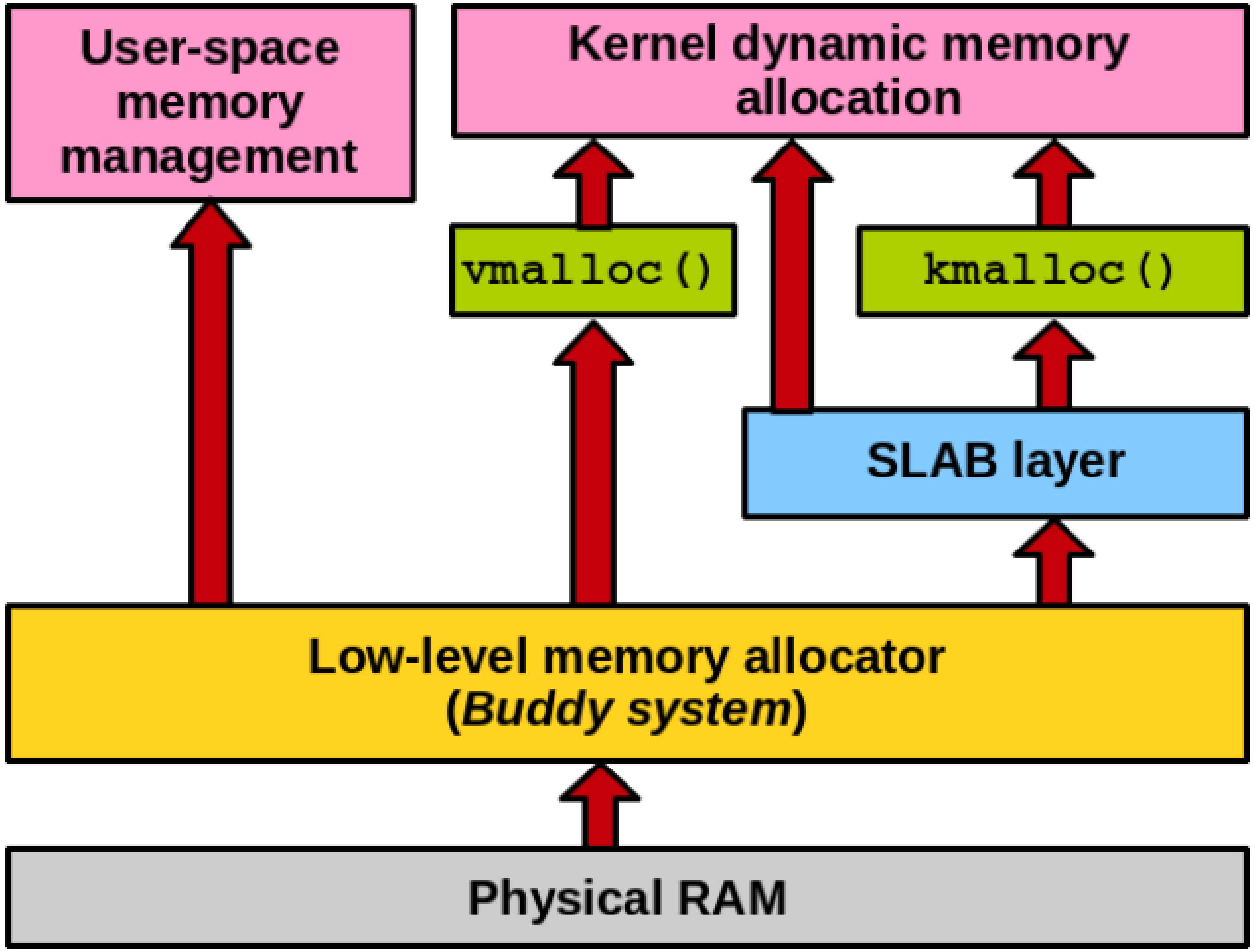


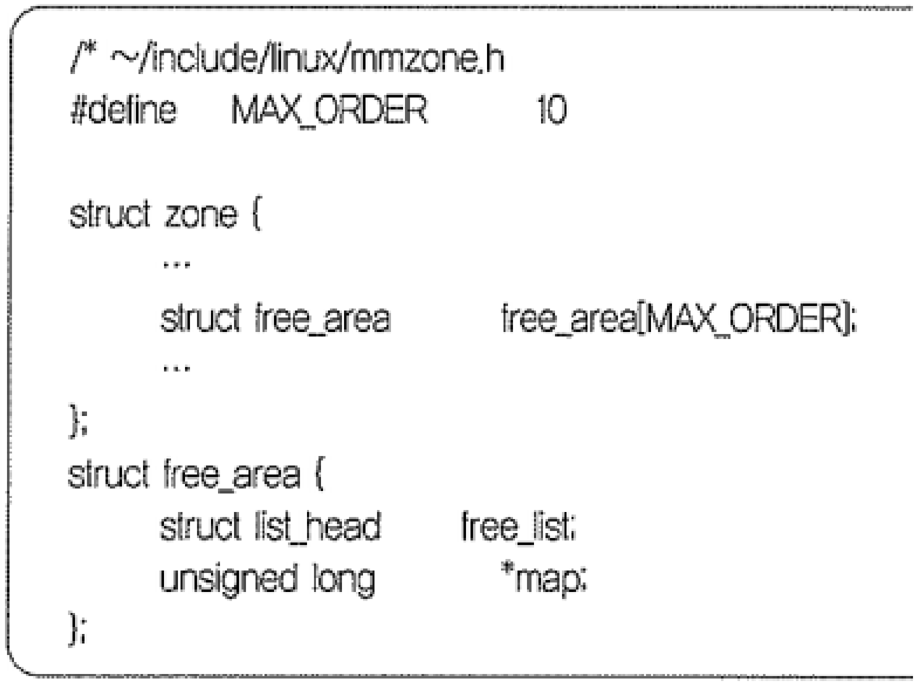
Figure 7.1. vmalloc Address Space

Buddy Allocator

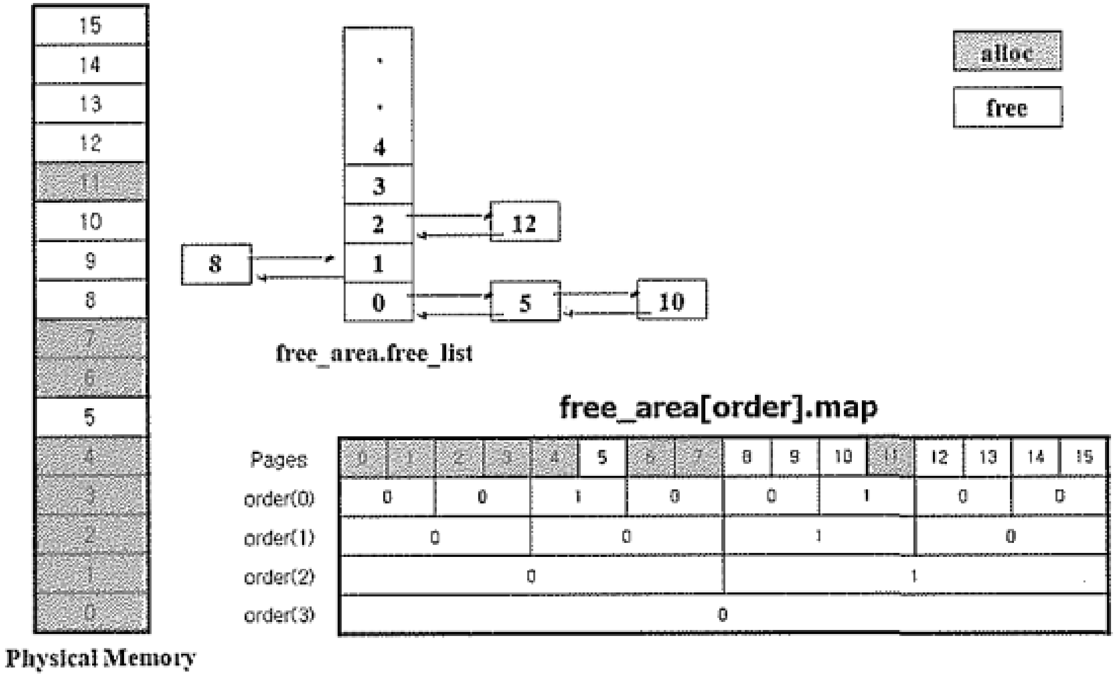

User-space memory management
Kernel dynamic memory allocation
vmalloc()
kmalloc()
SLAB layer
Low-level memory allocator
(*Buddy system*)
Physical RAM

Buddy Allocator

- Buddy Allocator mechanism



```
/* ~/include/linux/mmzone.h
#define    MAX_ORDER        10

struct zone {
        ...
        struct free_area          free_area[MAX_ORDER];
        ...
};
struct free_area {
        struct list_head      free_list;
        unsigned long           *map;
};
```

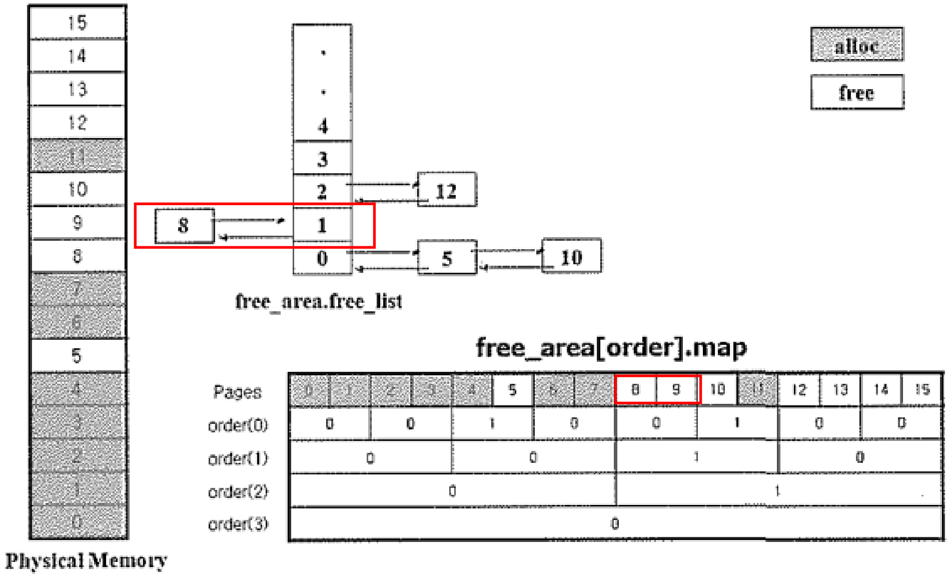


Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				1				0			
order(2)	0								1							
order(3)	0															

(1)

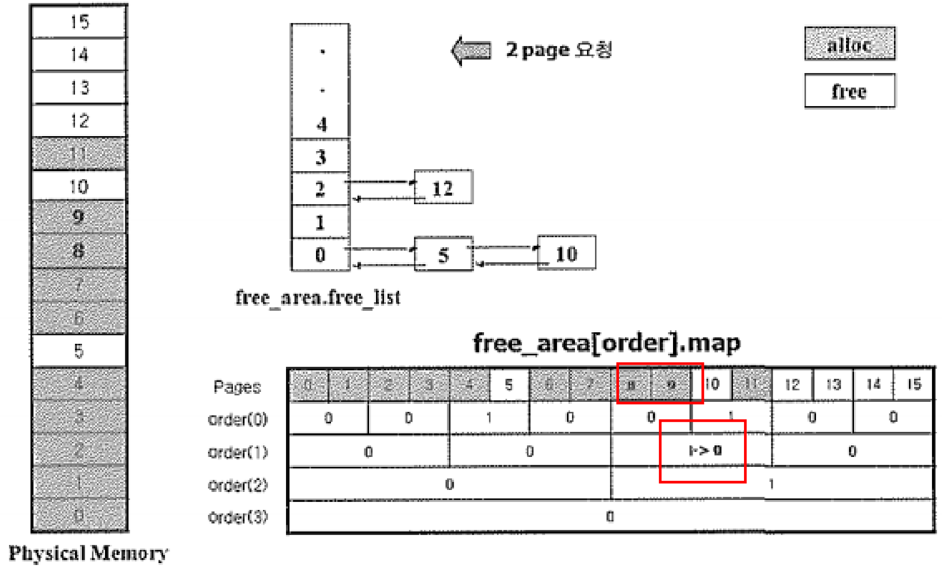
Buddy Allocator

- Buddy Allocator mechanism
  - On 2 pages are requested



Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				1				0			
order(2)	0								1							
order(3)	0															

(1)

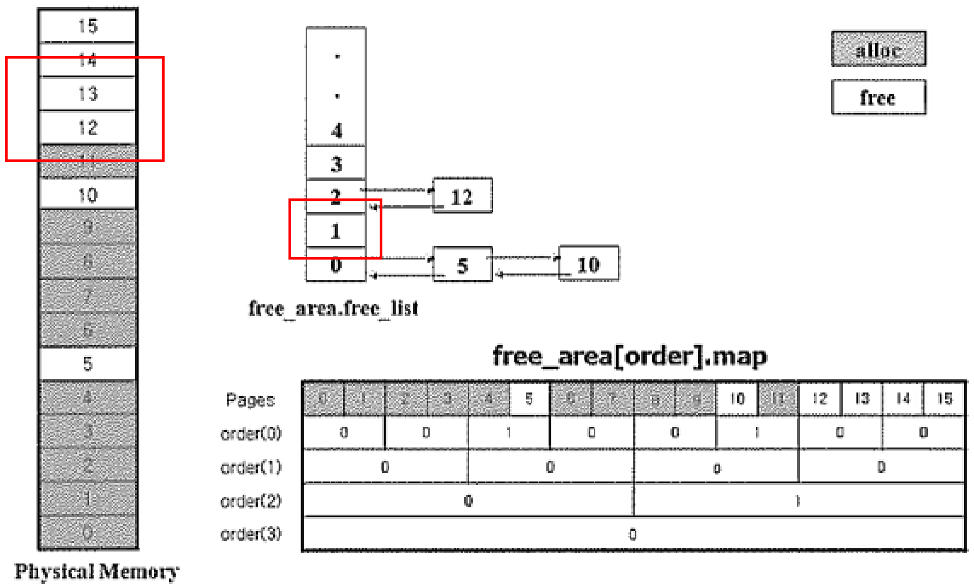


Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				1-> 0				0			
order(2)	0								1							
order(3)	0															

(2)

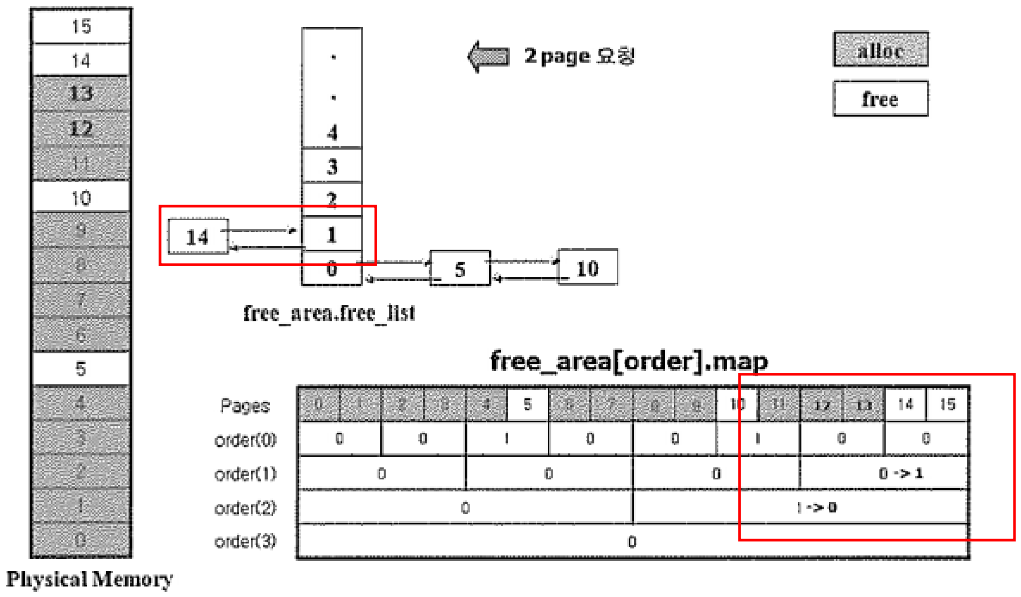
Buddy Allocator

- Buddy Allocator mechanism
  - On another 2 pages are requested



Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				o				0			
order(2)	0								1							
order(3)	0															

(1)

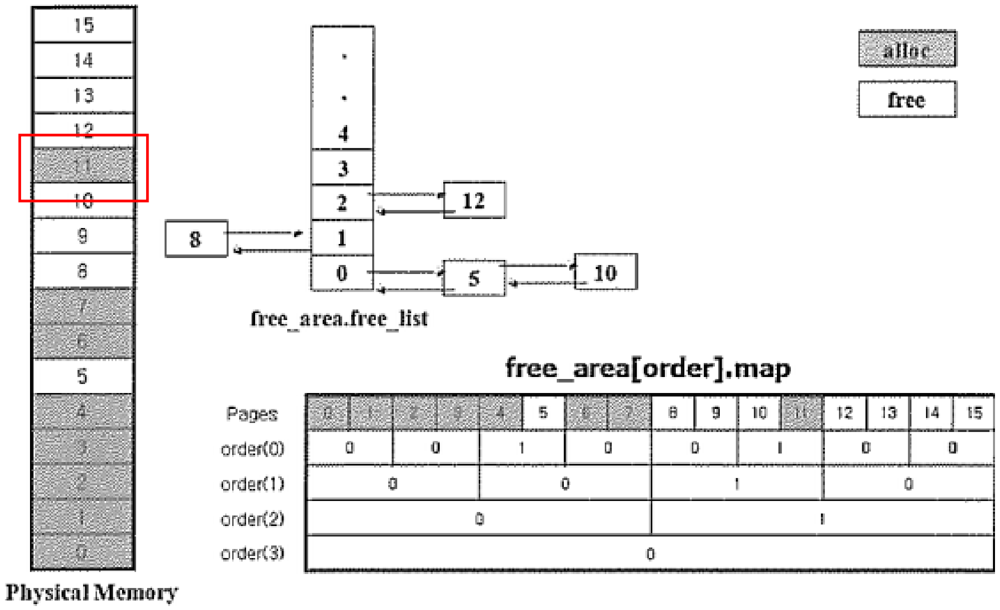


Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				0				0 -> 1			
order(2)	0								1 -> 0							
order(3)	0															

(2)

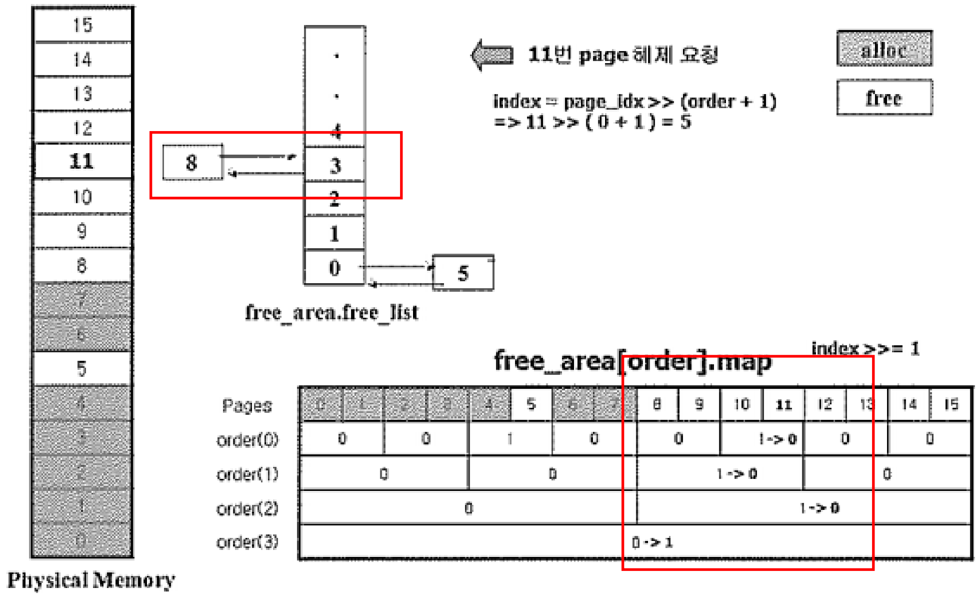
Buddy Allocator

- Buddy Allocator mechanism
  - On page 11 are freed



Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				1				0			
order(2)	0								1							
order(3)	0															

(1)

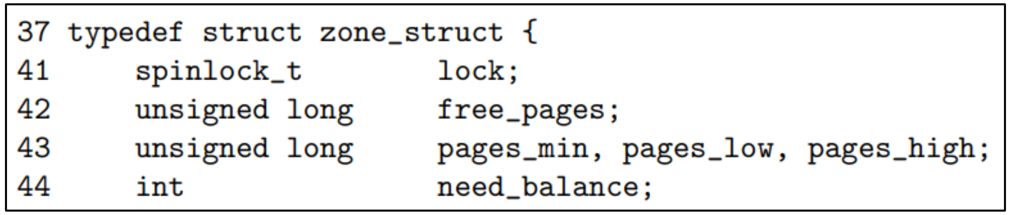


Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1		0		0		1 -> 0		0		0	
order(1)	0				0				1 -> 0				0			
order(2)	0								1 -> 0							
order(3)	0 -> 1															

(2)

Buddy Allocator

- Lazy Buddy allocator
  - Zone Watermarks



```
typedef struct zone_struct {
    spinlock_t        lock;
    unsigned long     free_pages;
    unsigned long     pages_min, pages_low, pages_high;
    int               need_balance;
```

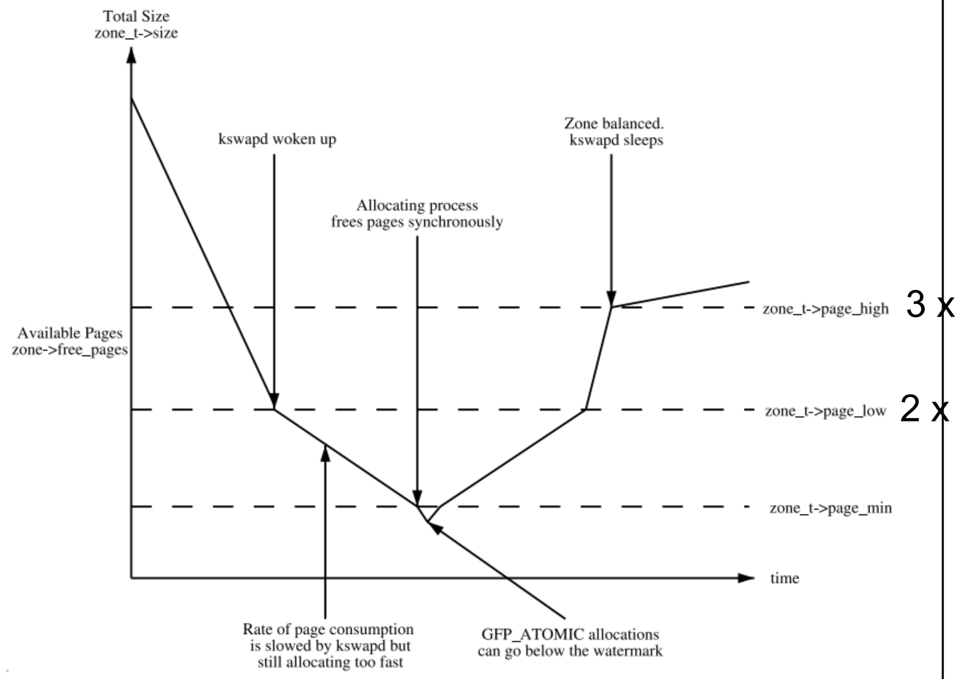
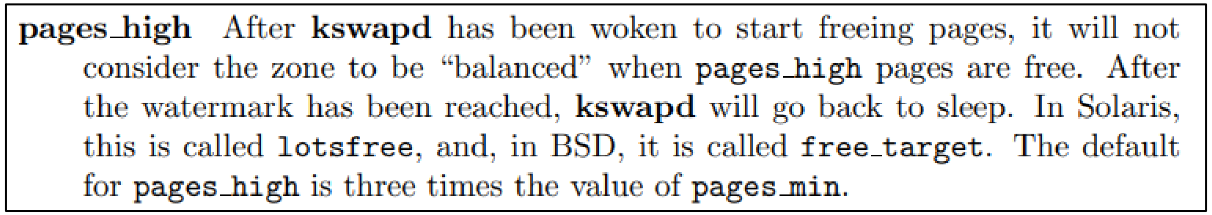


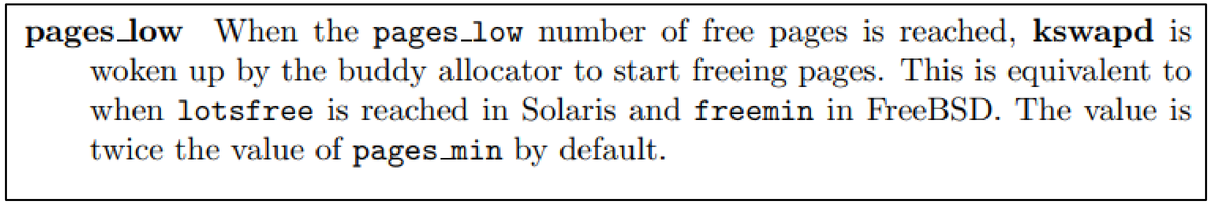
Figure 2.2. Zone Watermarks

3 x page_min

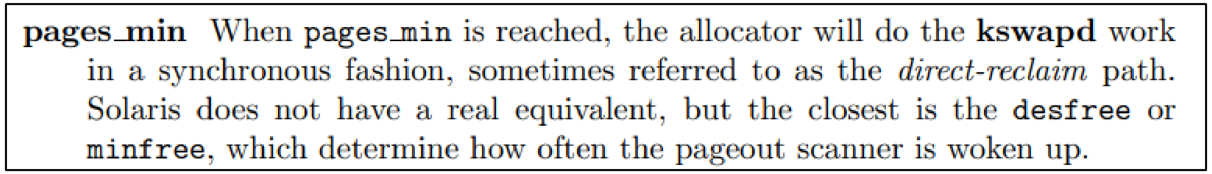
2 x page_min



pages_high After **kswapd** has been woken to start freeing pages, it will not consider the zone to be "balanced" when **pages_high** pages are free. After the watermark has been reached, **kswapd** will go back to sleep. In Solaris, this is called `lotsfree`, and, in BSD, it is called `free_target`. The default for **pages_high** is three times the value of **pages_min**.



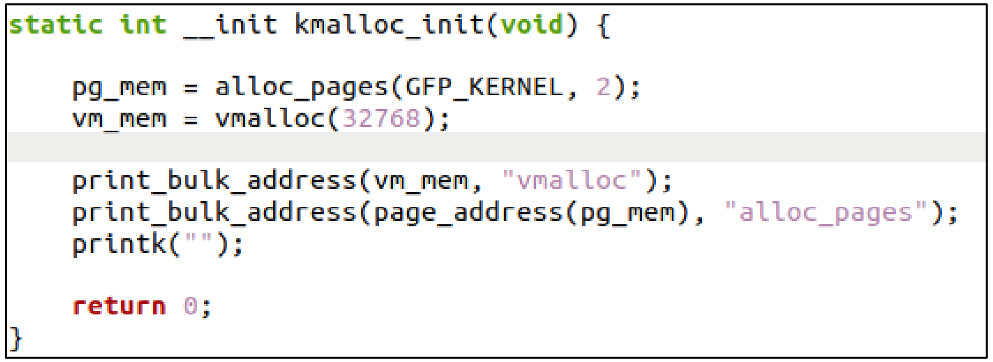
pages_low When the **pages_low** number of free pages is reached, **kswapd** is woken up by the buddy allocator to start freeing pages. This is equivalent to when `lotsfree` is reached in Solaris and `freemin` in FreeBSD. The value is twice the value of **pages_min** by default.



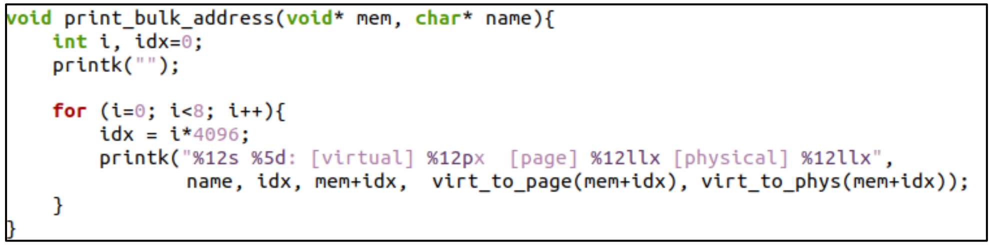
pages_min When **pages_min** is reached, the allocator will do the **kswapd** work in a synchronous fashion, sometimes referred to as the *direct-reclaim* path. Solaris does not have a real equivalent, but the closest is the `desfree` or `minfree`, which determine how often the pageout scanner is woken up.

Buddy Allocator

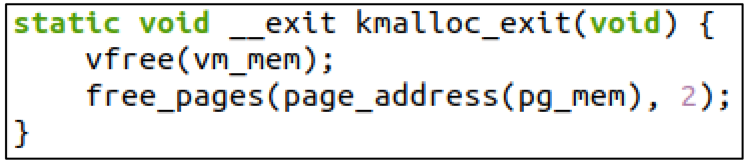
- alloc_pages vs vmalloc()



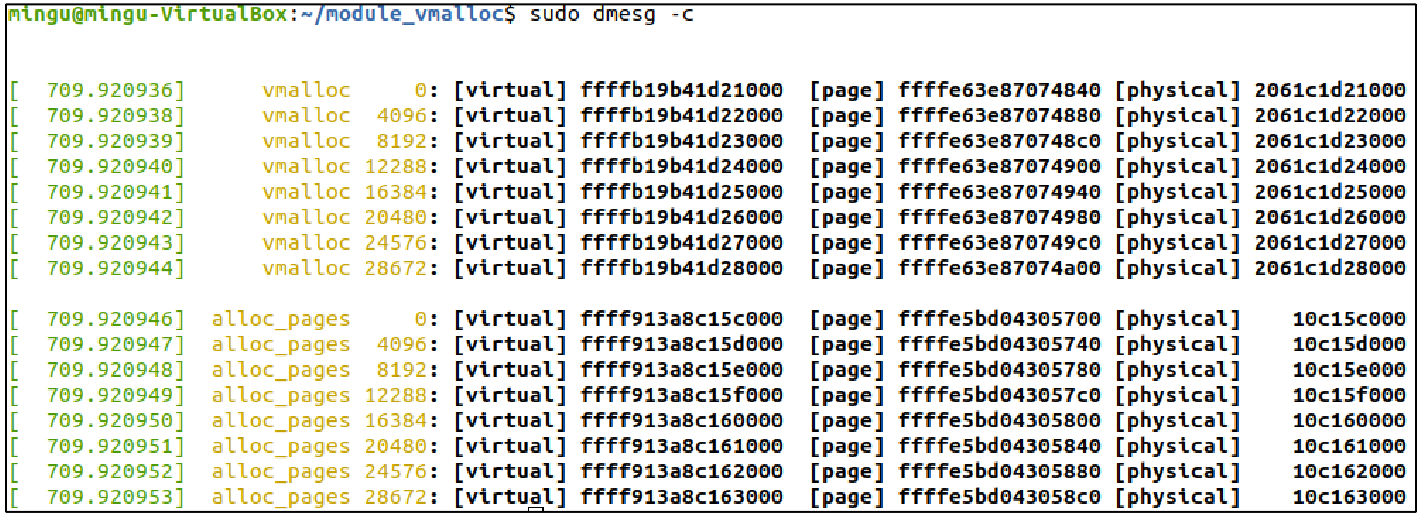
```
static int __init kmalloc_init(void) {

    pg_mem = alloc_pages(GFP_KERNEL, 2);
    vm_mem = vmalloc(32768);

    print_bulk_address(vm_mem, "vmalloc");
    print_bulk_address(page_address(pg_mem), "alloc_pages");
    printk("");

    return 0;
}
```



```
void print_bulk_address(void* mem, char* name){
    int i, idx=0;
    printk("");

    for (i=0; i<8; i++){
        idx = i*4096;
        printk("%12s %5d: [virtual] %12px  [page] %12llx [physical] %12llx",
                name, idx, mem+idx,  virt_to_page(mem+idx), virt_to_phys(mem+idx));
    }
}
```



```
static void __exit kmalloc_exit(void) {
    vfree(vm_mem);
    free_pages(page_address(pg_mem), 2);
}
```



```
mingu@mingu-VirtualBox:~/module_vmalloc$ sudo dmesg -c


[  709.920936]      vmalloc     0: [virtual] ffffb19b41d21000  [page] ffffe63e87074840 [physical] 2061c1d21000
[  709.920938]      vmalloc  4096: [virtual] ffffb19b41d22000  [page] ffffe63e87074880 [physical] 2061c1d22000
[  709.920939]      vmalloc  8192: [virtual] ffffb19b41d23000  [page] ffffe63e870748c0 [physical] 2061c1d23000
[  709.920940]      vmalloc 12288: [virtual] ffffb19b41d24000  [page] ffffe63e87074900 [physical] 2061c1d24000
[  709.920941]      vmalloc 16384: [virtual] ffffb19b41d25000  [page] ffffe63e87074940 [physical] 2061c1d25000
[  709.920942]      vmalloc 20480: [virtual] ffffb19b41d26000  [page] ffffe63e87074980 [physical] 2061c1d26000
[  709.920943]      vmalloc 24576: [virtual] ffffb19b41d27000  [page] ffffe63e870749c0 [physical] 2061c1d27000
[  709.920944]      vmalloc 28672: [virtual] ffffb19b41d28000  [page] ffffe63e87074a00 [physical] 2061c1d28000

[  709.920946]  alloc_pages     0: [virtual] ffff913a8c15c000  [page] ffffe5bd04305700 [physical]     10c15c000
[  709.920947]  alloc_pages  4096: [virtual] ffff913a8c15d000  [page] ffffe5bd04305740 [physical]     10c15d000
[  709.920948]  alloc_pages  8192: [virtual] ffff913a8c15e000  [page] ffffe5bd04305780 [physical]     10c15e000
[  709.920949]  alloc_pages 12288: [virtual] ffff913a8c15f000  [page] ffffe5bd043057c0 [physical]     10c15f000
[  709.920950]  alloc_pages 16384: [virtual] ffff913a8c160000  [page] ffffe5bd04305800 [physical]     10c160000
[  709.920951]  alloc_pages 20480: [virtual] ffff913a8c161000  [page] ffffe5bd04305840 [physical]     10c161000
[  709.920952]  alloc_pages 24576: [virtual] ffff913a8c162000  [page] ffffe5bd04305880 [physical]     10c162000
[  709.920953]  alloc_pages 28672: [virtual] ffff913a8c163000  [page] ffffe5bd043058c0 [physical]     10c163000
```

- alloc_pages()
  - contiguous VA, contiguous PA
- vmalloc()
  - contiguous VA, contiguous PA

Buddy Allocator

- RACE CONDITION: Alloc_pages vs vmalloc()

While(1)

vmalloc(random()/32*PAGE_SIZE)
vfree()

alloc_pages(GFP_KERNEL, random()%4)
free_pages()

malloc(random()/64*PAGE_SIZE)
~~free()~~

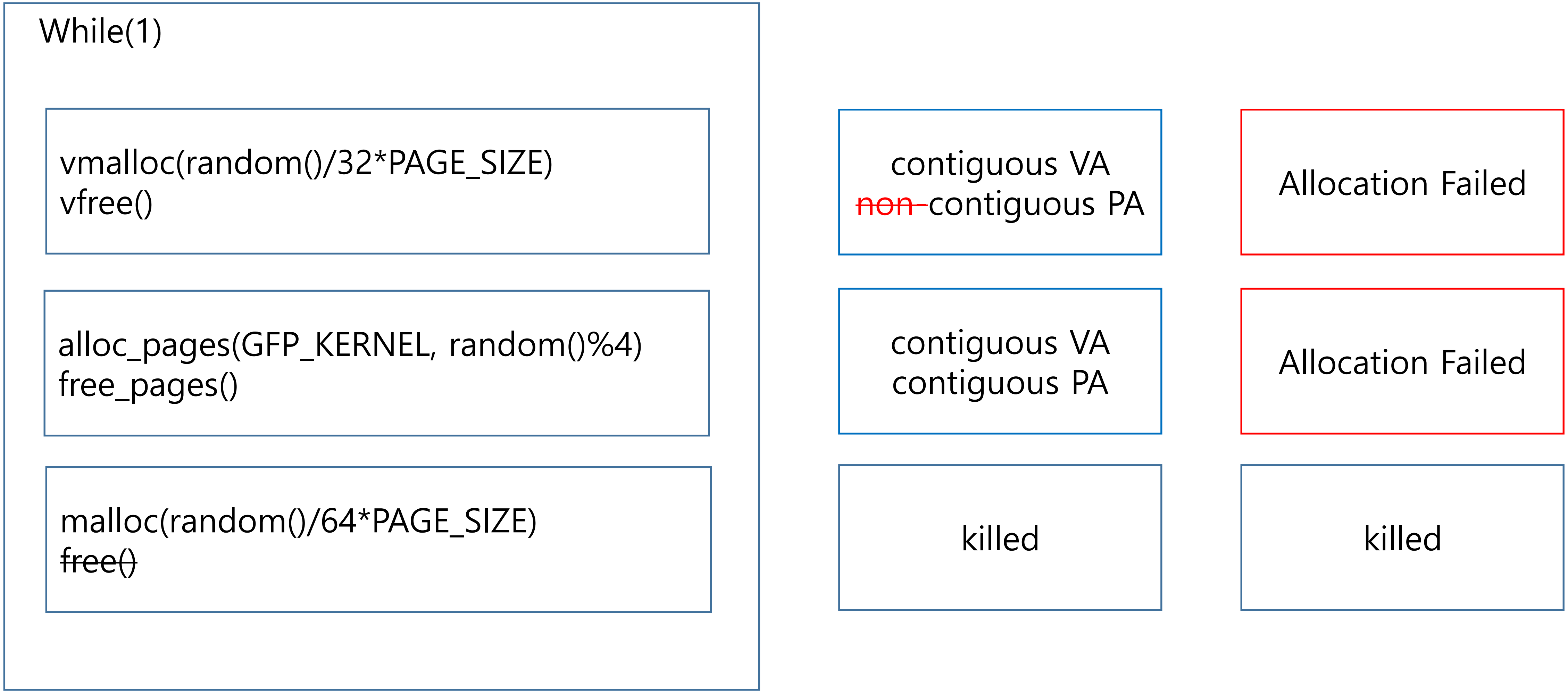
contiguous VA
non-contiguous PA

contiguous VA
contiguous PA

killed

Buddy Allocator

- RACE CONDITION: Alloc_pages vs vmalloc()



While(1)

vmalloc(random()/32*PAGE_SIZE)
vfree()

alloc_pages(GFP_KERNEL, random()%4)
free_pages()

malloc(random()/64*PAGE_SIZE)
~~free()~~

contiguous VA
~~non~~ contiguous PA

Allocation Failed

contiguous VA
contiguous PA

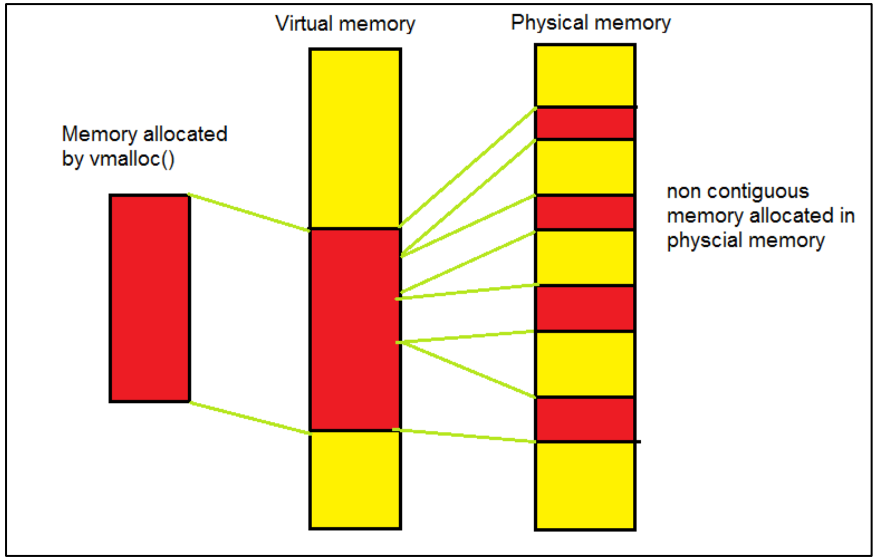
Allocation Failed

killed

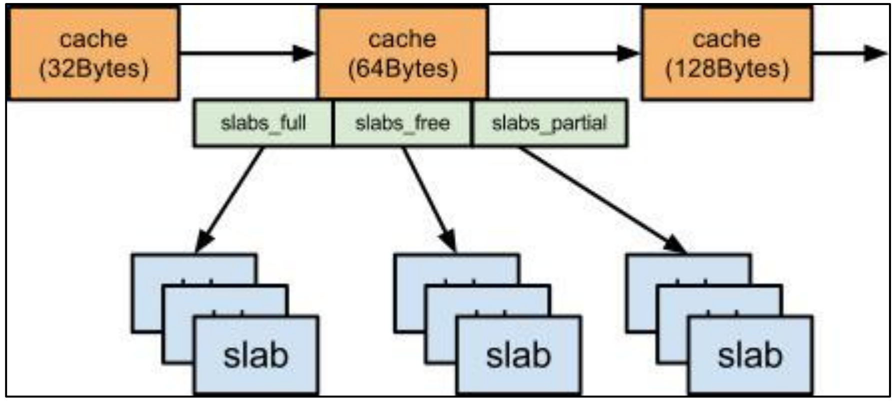
killed

Slab Allocator

- 커널 내부에서 가장 빈번한 작업
  - 자료구조(struct task, struct …) 할당/해제
  - 이를 vmalloc으로 할당/해제 ?
    - ✓ Memory fragmentation
    - ✓ Allocate & Free overhead
    - ✓ Non-continguous PA <-> continguous VA
    - ✓ Need continguous PA for hardware device
      - 가상 주소 처리하는 메모리 장치 x

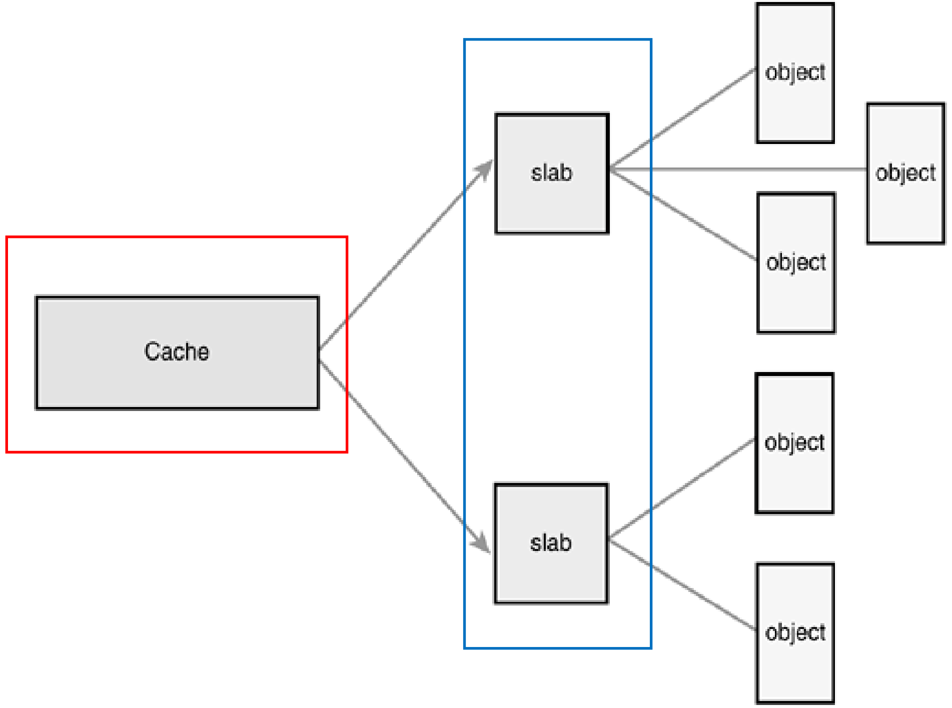

Virtual memory
Physical memory
Memory allocated by vmalloc()
non contiguous memory allocated in physcial memory

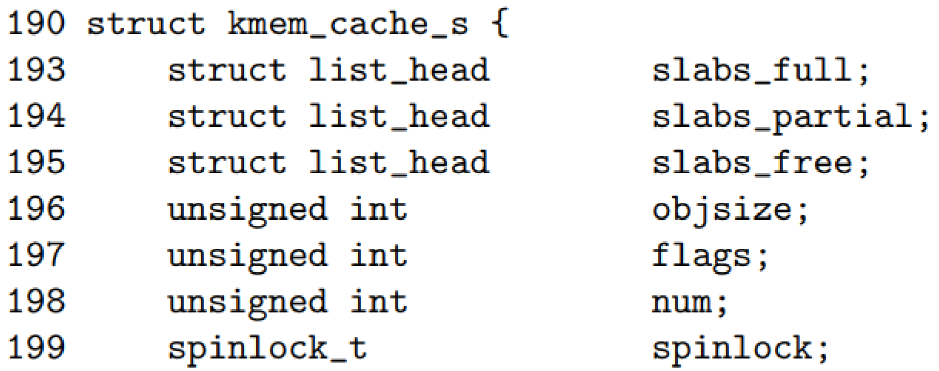
- 대신 Free list를 사용
  - 자료구조 사용 이후, 해제하지 않고 free_list에 추가
  - 일종의 객체 Cache
  - But…
    - ✓ 메모리 부족 시, 어떤 캐시 크기를 줄일 것일지 모름
      -> Slab Allocator 사용하여, 캐시 관리



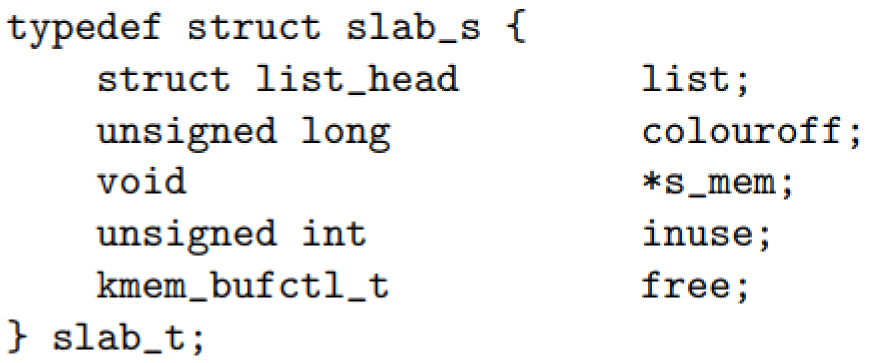
Slab Allocator

- Slab Allocator
  - Cache – Slab – Object
  - Slab
    - ✓ full / partial / free
    - ✓ allocate on partial -> free -> new
      - Memory fragmentation mitigate





```
struct kmem_cache_s {
        struct list_head         slabs_full;
        struct list_head         slabs_partial;
        struct list_head         slabs_free;
        unsigned int             objsize;
        unsigned int             flags;
        unsigned int             num;
        spinlock_t               spinlock;
```

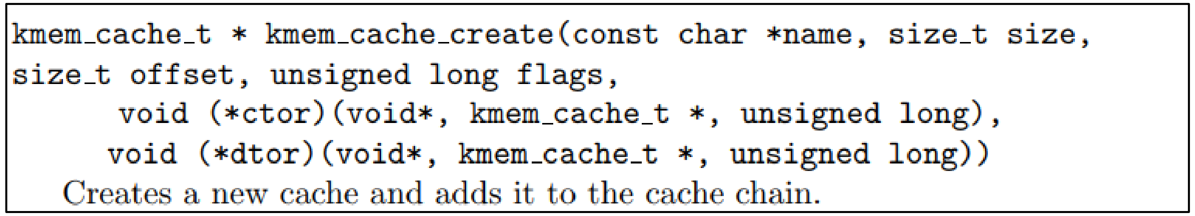


```
typedef struct slab_s {
    struct list_head        list;
    unsigned long           colouroff;
    void                    *s_mem;
    unsigned int            inuse;
    kmem_bufctl_t           free;
} slab_t;
```

Slab Allocator

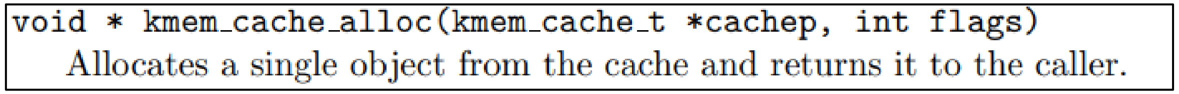
- kmalloc() / kfree()
  - 바이트 단위로 할당/해제
  - Contiguous VA, contiguous PA

- Kmem_cache
  - Kmem_cache_create()
  - Kmem_cache_alloc() <- kmalloc()
  - Kmem_cache_free() <- kfree()
  - Kmem_cache_destroy()



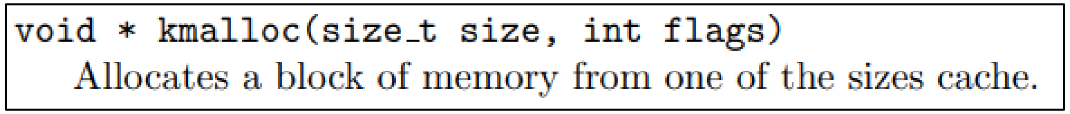
```
kmem_cache_t * kmem_cache_create(const char *name, size_t size,
size_t offset, unsigned long flags,
      void (*ctor)(void*, kmem_cache_t *, unsigned long),
     void (*dtor)(void*, kmem_cache_t *, unsigned long))
```

Creates a new cache and adds it to the cache chain.



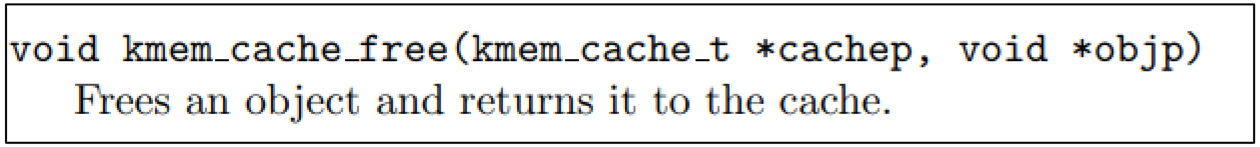
```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

Allocates a single object from the cache and returns it to the caller.



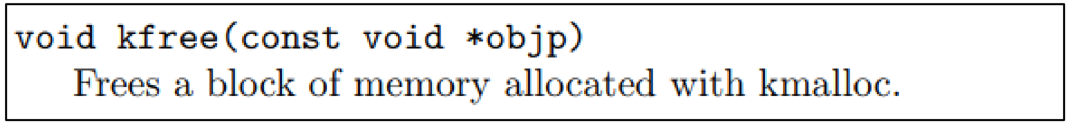
```
void * kmalloc(size_t size, int flags)
```

Allocates a block of memory from one of the sizes cache.



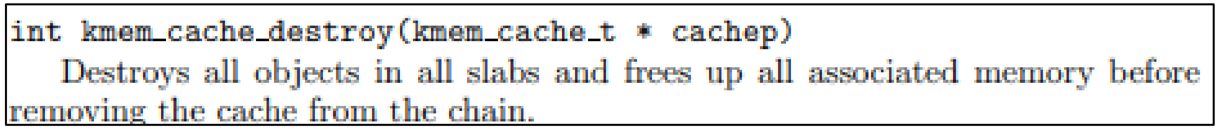
```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

Frees an object and returns it to the cache.



```
void kfree(const void *objp)
```

Frees a block of memory allocated with kmalloc.

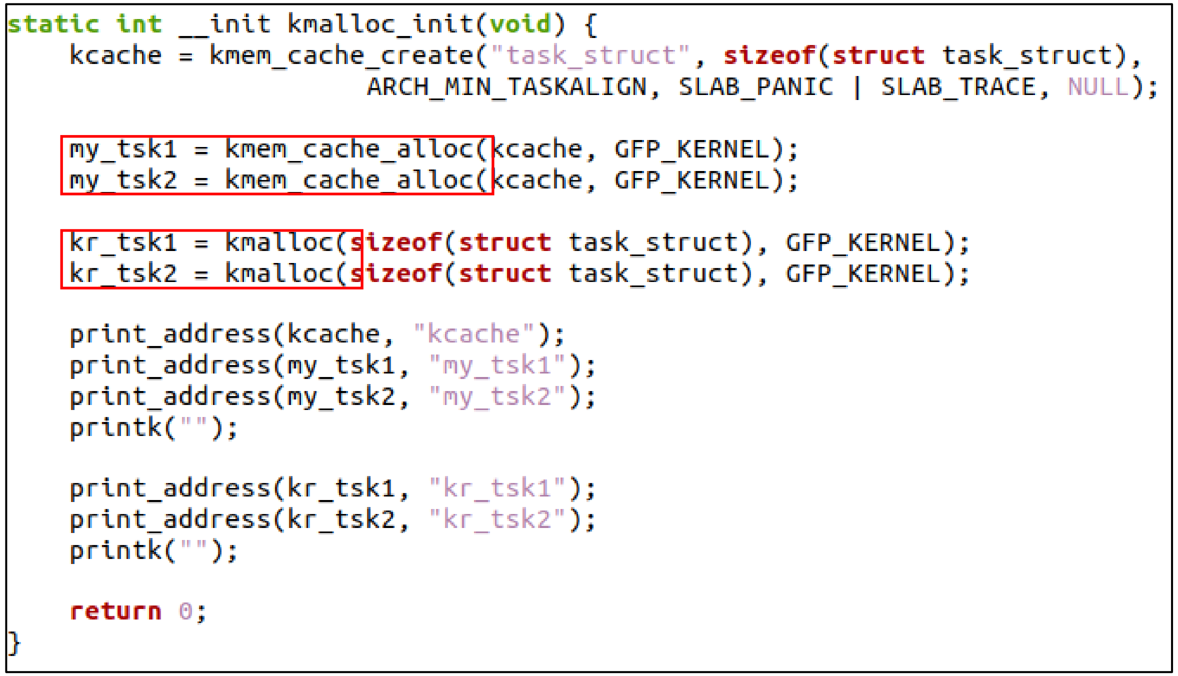


```
int kmem_cache_destroy(kmem_cache_t * cachep)
```

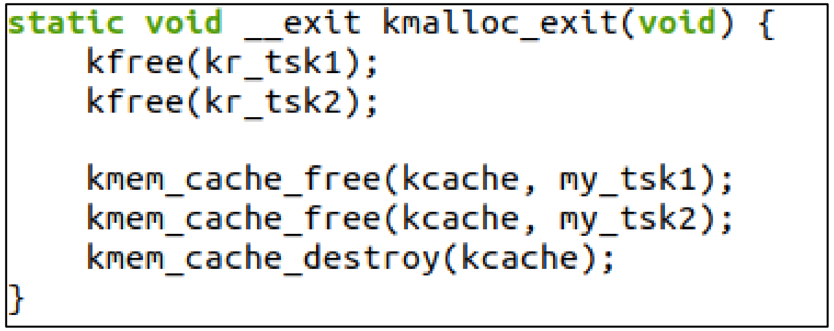
Destroys all objects in all slabs and frees up all associated memory before removing the cache from the chain.

Slab Allocator

- kmem_cache_alloc() vs kmalloc()
- kmem_cache_free() vs kfree()



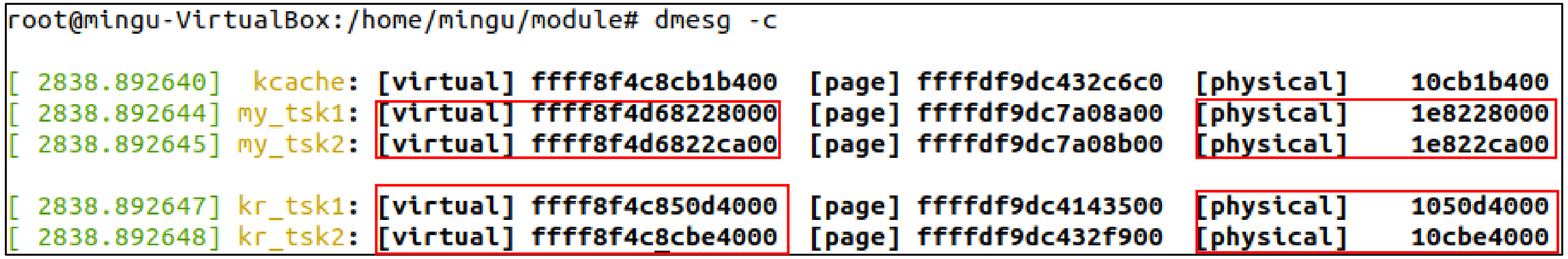
```
static int __init kmalloc_init(void) {
    kcache = kmem_cache_create("task_struct", sizeof(struct task_struct),
                               ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_TRACE, NULL);

    my_tsk1 = kmem_cache_alloc(kcache, GFP_KERNEL);
    my_tsk2 = kmem_cache_alloc(kcache, GFP_KERNEL);

    kr_tsk1 = kmalloc(sizeof(struct task_struct), GFP_KERNEL);
    kr_tsk2 = kmalloc(sizeof(struct task_struct), GFP_KERNEL);

    print_address(kcache, "kcache");
    print_address(my_tsk1, "my_tsk1");
    print_address(my_tsk2, "my_tsk2");
    printk("");

    print_address(kr_tsk1, "kr_tsk1");
    print_address(kr_tsk2, "kr_tsk2");
    printk("");

    return 0;
}
```



```
static void __exit kmalloc_exit(void) {
    kfree(kr_tsk1);
    kfree(kr_tsk2);

    kmem_cache_free(kcache, my_tsk1);
    kmem_cache_free(kcache, my_tsk2);
    kmem_cache_destroy(kcache);
}
```

- kmem_cache_alloc()
  - module
  - close VA & PA

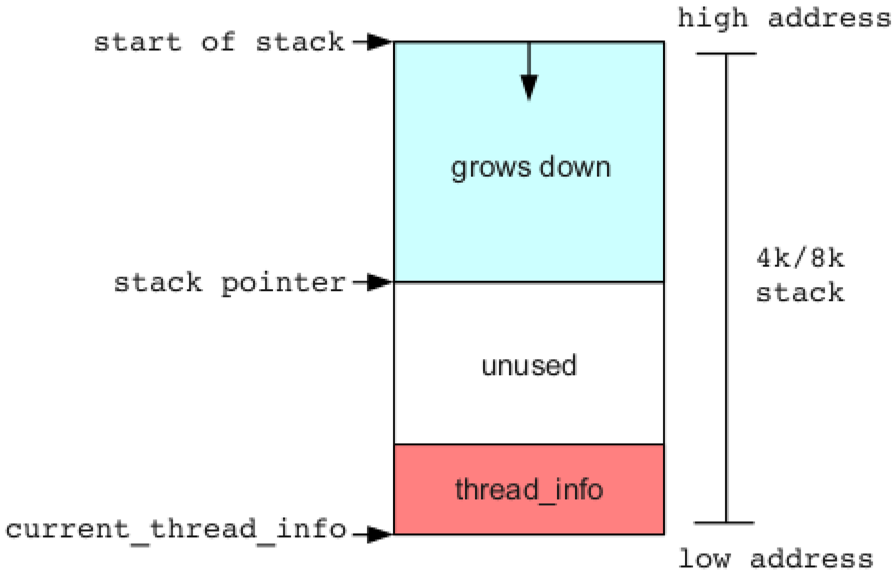
- kmalloc()
  - Slab
  - Far VA & PA



```
root@mingu-VirtualBox:/home/mingu/module# dmesg -c

[ 2838.892640]  kcache: [virtual] ffff8f4c8cb1b400  [page] ffffdf9dc432c6c0  [physical]    10cb1b400
[ 2838.892644] my_tsk1: [virtual] ffff8f4d68228000  [page] ffffdf9dc7a08a00  [physical]    1e8228000
[ 2838.892645] my_tsk2: [virtual] ffff8f4d6822ca00  [page] ffffdf9dc7a08b00  [physical]    1e822ca00

[ 2838.892647] kr_tsk1: [virtual] ffff8f4c850d4000  [page] ffffdf9dc4143500  [physical]    1050d4000
[ 2838.892648] kr_tsk2: [virtual] ffff8f4c8cbe4000  [page] ffffdf9dc432f900  [physical]    10cbe4000
```

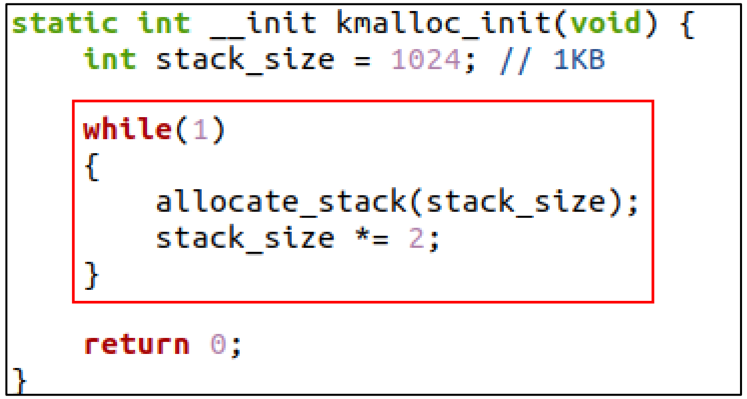
Kernel Stack

- Stack
  - User Process: big, dynamic
  - Kernel: small, fixed
    - ✓ minimize mermory usage
    - ✓ do no not manage kernel stack code
- Kernel Stack size
  - 2 * Page size
    - ✓ 32bit: 8KB, 64bit: 16KB
- Kernel Stack overflow
  - therad_info

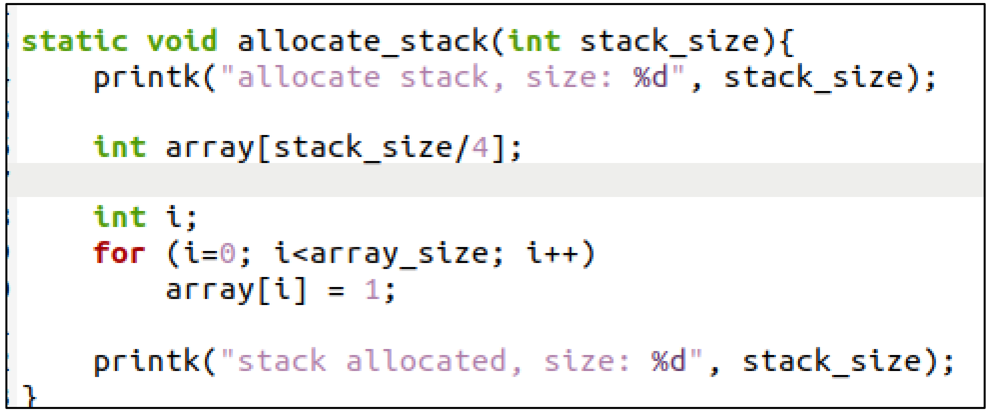


Kernel Stack

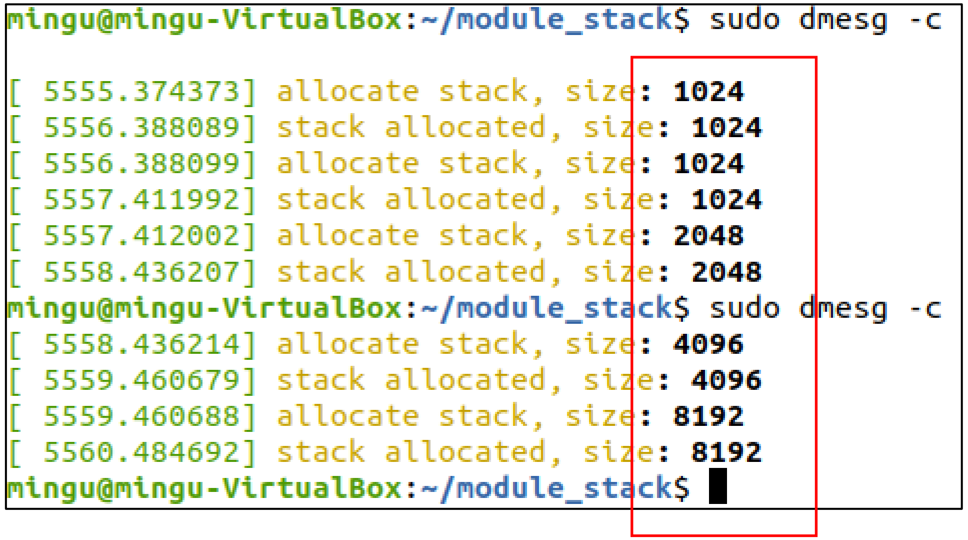
- Kernel Stack Overflow



```
static int __init kmalloc_init(void) {
    int stack_size = 1024; // 1KB

    while(1)
    {
        allocate_stack(stack_size);
        stack_size *= 2;
    }

    return 0;
}
```



```
static void allocate_stack(int stack_size){
    printk("allocate stack, size: %d", stack_size);

    int array[stack_size/4];

    int i;
    for (i=0; i<array_size; i++)
        array[i] = 1;

    printk("stack allocated, size: %d", stack_size);
}
```



```
mingu@mingu-VirtualBox:~/module_stack$ sudo dmesg -c

[ 5555.374373] allocate stack, size: 1024
[ 5556.388089] stack allocated, size: 1024
[ 5556.388099] allocate stack, size: 1024
[ 5557.411992] stack allocated, size: 1024
[ 5557.412002] allocate stack, size: 2048
[ 5558.436207] stack allocated, size: 2048
mingu@mingu-VirtualBox:~/module_stack$ sudo dmesg -c
[ 5558.436214] allocate stack, size: 4096
[ 5559.460679] stack allocated, size: 4096
[ 5559.460688] allocate stack, size: 8192
[ 5560.484692] stack allocated, size: 8192
mingu@mingu-VirtualBox:~/module_stack$ 
```

system doesn't work -> reboot

- Stack Size = 2 * Page_Size =16KB

- Allocate array[16KB]
  - Kernel stack over flow

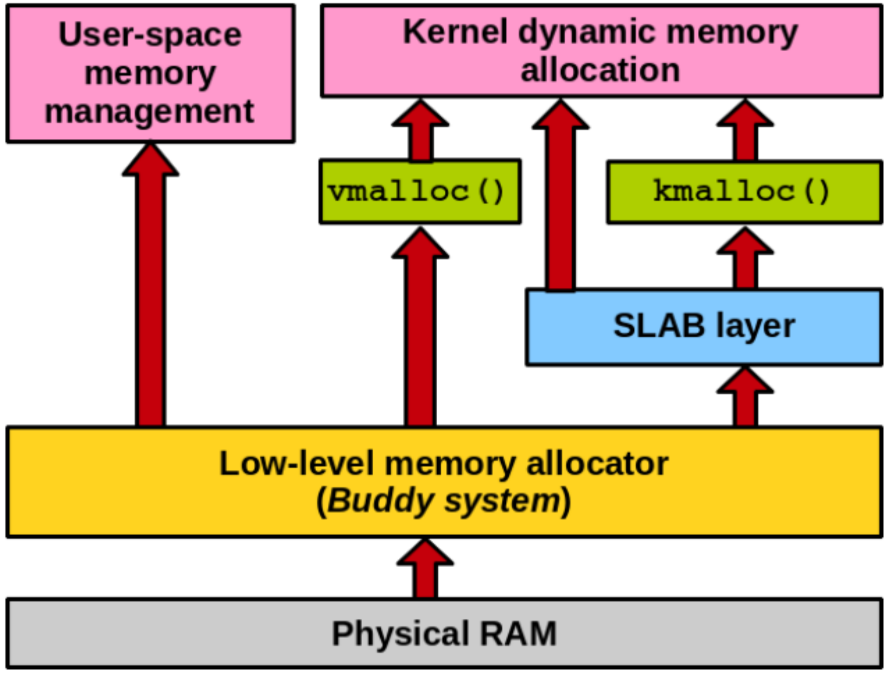
Q & A

- Physical Memory Data Structure
  - UMA/NUMA
  - Node, Zone, Page

- Buddy Allocator
  - alloc_pages / free_pages()
  - vmalloc() / vfree()
  - practice1: alloc_pages() vs vmalloc()

- Slab Allocator
  - kmalloc() / kfree()
  - practice2: kmem_cache_alloc() vs kmalloc()

- Kernel Stack
  - practice3: kernel size


User-space memory management
Kernel dynamic memory allocation
vmalloc()
kmalloc()
SLAB layer
Low-level memory allocator
(*Buddy system*)
Physical RAM