BUDDY SYSTEM & SLAB ALLOCATOR

주요 함수 코드 해석

2007.12.15 1ST EDITION

김기오 (*gurugio@gmail_nospam.com***)**

http://www.asmlove.co.kr

목차

1. kmem_cache_create

linux 2.4.19

이 함수는 새로운 cache를 만들고 cache_chain에 연결하는 일을 한다. cache_cache slab cache에서 kmem_cache_t 를 할당받는다.

```
/**
 * kmem_cache_create - Create a cache.
 * @name: A string which is used in /proc/slabinfo to identify this cache.
 * @size: The size of objects to be created in this cache.
 * @offset: The offset to use within the page.
 * @flags: SLAB flags
 * @ctor: A constructor for the objects.
 * @dtor: A destructor for the objects.
 *
 * Returns a ptr to the cache on success, NULL on failure.
 * Cannot be called within a int, but can be interrupted.
 * The @ctor is run when new pages are allocated by the cache
 * and the @dtor is run before the pages are handed back.
 * The flags are
 *
 * %SLAB_POISON - Poison the slab with a known test pattern (a5a5a5a5)
```

```
 * to catch references to uninitialised memory.
 *
 * %SLAB_RED_ZONE - Insert `Red' zones around the allocated memory to check
 * for buffer overruns.
 *
 * %SLAB_NO_REAP - Don't automatically reap this cache when we're under
 * memory pressure.
 *
 * %SLAB_HWCACHE_ALIGN - Align the objects in this cache to a hardware
 * cacheline.  This can be beneficial if you're counting cycles as closely
 * as davem.
 */
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))
{
```

함수 인자에 대한 설명은 소스 주석에 나와있는 그대로이다.

```
        const char *func_nm = KERN_ERR "kmem_create: ";
        size_t left_over, align, slab_size;
        kmem_cache_t *cachep = NULL;

        /* Get cache's description obj. */
        cachep = (kmem_cache_t *) kmem_cache_alloc(&cache_cache, SLAB_KERNEL);
        if (!cachep)
                goto opps;
        memset(cachep, 0, sizeof(kmem_cache_t));
```

cache_cache에서 cache를 위한 kmem_cache_t 데이터를 얻어온다. kmem_cache_init()함수에서는 cache_cache를 위한 페이지를 할당받지 않았으므로 이 함수가 처음 호출되었을 때는 cache_cache가 아무런 cache도 가지지 않을 때가 된다. 이렇게 해당 cache에 사용가능한 object가 없을 때 kmem_cache_alloc()은 kmem_cache_grow()를 호출하여 페이지를 할당받고 페이지에 slab를 설정하고 해당 cache에 맞는 object들을 초기화해서 object를 반환해준다.

```
        /* Check that size is in terms of words.  This is needed to avoid
         * unaligned accesses for some archs when redzoning is used, and makes
         * sure any on-slab bufctl's are also correctly aligned.
         */
        if (size & (BYTES_PER_WORD-1)) {
```

```
            size += (BYTES_PER_WORD-1);
            size &= ~(BYTES_PER_WORD-1);
            printk("%sForcing size word alignment - %s₩n", func_nm, name);
    }
```

예를 들어 object의 크기가 31바이트였다고 하면 펜티엄 환경에서는 BYTES_PER_WORD가 4이므로 word의 경계에 맞지 안게된다. 이 코드는 4바이트 경계에 맞게 object의 크기를 32바이트로 맞춰주는 일을 한다.

size = 31이므로 0001 1111이다. BYTES_PER_WROD-1은 3이므로 0001 1111 & 0000 0011 은 0이 아니므로 size에 3을 더하고 ~(BYTES_PER_WORD-1) = 1111 1100 과 AND를 해서 32로 만든다.

```
    align = BYTES_PER_WORD;
    if (flags & SLAB_HWCACHE_ALIGN)
            align = L1_CACHE_BYTES;
```

하드웨어 캐쉬 경계를 맞춘다. 펜티엄에서 BYTE_PER_WORD은 4이고 L1 하드웨어 캐쉬의 경계는 32바이트이다. 리눅스에서는 시스템에 하드웨어 캐쉬가 유일하다고 가정하므로 slab를 하드웨어 캐쉬에 맞게 생성하도록 플래그를 지정하면 L1 하드웨어 캐쉬의 경계에 맞게 32바이트 경계를 사용한다.

```
    /* Determine if the slab management is 'on' or 'off' slab. */
    if (size >= (PAGE_SIZE>>3))
            /*
             * Size is large, assume best to place the slab management obj
             * off-slab (should allow better packing of objs).
             */
            flags |= CFLGS_OFF_SLAB;
```

slab를 페이지 내부에 둘건지 밖에 둘건지를 결정해야한다. object의 크기가 작으면 한 페이지 안에 여러개의 object를 둘 수있고 거기에 slab까지 저장할 수 있지만 만약 크기가 크면 효율성이 떨어지게된다.

만약 object가 2048바이트라면 4096바이트의 페이지에 두개의 object가 들어갈 수 있는데 만약 약 20바이트 크기의 slab을 페이지 안에 두면 하나의 페이지에 하나의 object와 하나의 slab를 저장하므로 internal fragmentation이 커지게 된다. 이런 경우에는 slab를 페이지 밖에 따로 관리하는 방법이 사용되게 된다.

여기서는 페이지 크기의 1/2^3=1/8배, 즉 4K/8=512byte보다 큰 object를 위한 슬랩은 외부 슬랩으로 만든다.

```
        if (flags & SLAB_HWCACHE_ALIGN) {
                /* Need to adjust size so that objs are cache aligned. */
                /* Small obj size, can get at least two per cache line. */
                /* FIXME: only power of 2 supported, was better */
                while (size < align/2)
                        align /= 2;
                size = (size+align-1)&(~(align-1));
        }
```

cache의 플래그에 하드웨어 캐쉬 플래그를 지정되어 있다면 align의 크기가 L1_CACHE_BYTES의 값인 32가 되었을 것이다. 그리고 플래그에 상관없이 size는 BYTES_PER_WORD의 배수가 됐을 것이다. 이 때 object의 크기가 align 크기의 절반보다 작으면 align을 줄여나간다. 만약 size가 8이라면 align을 16으로 줄여서 사용하게된다. 그리고 바뀐 align으로 size의 경계를 맞춘다. 결국 align 값도 16이 되고 size 값도 16이 된다. 이것은 하드웨어 캐쉬에 2개의 object를 정렬시킨 것이 된다.

이 코드의 의미는 하드웨어 캐쉬에 object가 2개 들어갈 수 있는지 확인해보고 그렇다면 4개, 8개가 들어갈 수 있는지 확인해서 최대한 많이 들어갈 수 있게 size를 맞추는 것이다. 주의할 것은 size가 최초 align의 크기 32의 반, 16보다 작을 때만 의미가 있다는 것이다. 만약 size가 16이상이라면 size는 size = (size+align-1)&(~(align-1)); 에 의해 32가 될 것이다.

```
        /* Cal size (in pages) of slabs, and the num of objs per slab.
         * This could be made much more intelligent.  For now, try to avoid
         * using high page-orders for slabs.  When the gfp() funcs are more
         * friendly towards high-order requests, this should be changed.
         */
        do {
                unsigned int break_flag = 0;
cal_wastage:
                kmem_cache_estimate(cachep->gfporder, size, flags,
                                    &left_over, &cachep->num);
```

cache에 할당된 페이지 안에 몇 개의 object가 들어갈 수 있는지 (cachep->num) 페이지의 끝에 남는 공간이 얼마인지 (left_over) 계산한다.

```
                if (break_flag)
                        break;
                if (cachep->gfporder >= MAX_GFP_ORDER)
                        break;
```

```
if (!cachep->num)
        goto next;
if (flags & CFLGS_OFF_SLAB && cachep->num > offslab_limit) {
        /* Oops, this num of objs will cause problems. */
        cachep->gfporder--;
        break_flag++;
        goto cal_wastage;
}
```

에러 검사 부분이다. MAX_GFP_ORDER는 5이다. 즉 한 cache에 5페이지 이상을 할달할 수 없게되어있다.

cachep->num 이 0이면 할달된 페이지가 너무 작아서 object를 저장할 수 없다는 것이다. next 라벨로 점프해서 페이지를 하나 더 늘려가면서 다시 kmem_cache_estimate() 함수를 실행하게 된다.

offslab_limit 변수는 이전 장에서 설명한 대로 외부 슬랩을 사용할 때 한 슬랩 디스크립터가 관리할 수 있는 object의 최대 갯수를 말한다. 만약 이 한계를 넘어서게 되면 cache에 할당된 페이지를 줄여서 다시 kmem_cache_estimate() 함수를 시도하게 된다. 할당된 페이지를 줄이면 한 슬랩 디스크립터가 관리해야 할 object의 수도 줄어들게 된다.

break_flag는 외부 슬랩이 사용될 때 단 한번만 페이지 order를 줄이도록 하기위해서 사용된다.

```
/*
 * The buddy Allocator will suffer if it has to deal with too many
* allocators of a large order. So while large numbers of objects is
* good, large orders are not so slab_break_gfp_order forces a balance
 */
if (cachep->gfporder >= slab_break_gfp_order)
        break;
```

주석에 써진 그대로 할당된 페이지가 slab_break_gfp_order (보통 1) 이상이면 페이지의 갯수를 늘리지 않고 그대로 루프를 끝내고 다음으로 넘어간다.

```
if ((left_over*8) <= (PAGE_SIZE<<cachep->gfporder))
        break;   /* Acceptable internal fragmentation. */
```

internal fragmentation의 크기에 대해서 대략적으로 체크해본다. 슬랩에 할당된 메모리 크기의 1/8 이하이면 적당하다고 생각하고 루프를 끝낸다.

```
next:
                cachep->gfporder++;
        } while (1);
```

페이지를 늘려서 다시 시도한다. 방금 체크한 조건들대로 페이지를 늘려서 슬랩이 관리할 수 있는 object의 갯수와 내부 단편화의 크기등의 조건을 고려했을 때 페이지의 갯수를 늘릴 필요가 있다는 판단을 했다는 뜻이다.

결국 이 루프는 하나의 슬랩에 할당되는 페이지의 갯수와 슬랩당 처리할 수 있는 object의 갯수, 외부 슬랩의 조건들, internal fragmentation의 크기등 cache가 가져야 하는 속성들에 대해서 검사하고 효율을 맞추는 과정이 된다.

```
        if (!cachep->num) {
                printk("kmem_cache_create: couldn't create cache %s.₩n", name);
                kmem_cache_free(&cache_cache, cachep);
                cachep = NULL;
                goto opps;
        }
```

object가 cache에 저장되기에는 너무 크면 cache_cache에서 얻었던 캐시를 지우고 NULL을 반환한다.

```
        slab_size =
L1_CACHE_ALIGN(cachep->num*sizeof(kmem_bufctl_t)+sizeof(slab_t));
```

슬랩의 크기는 슬랩 자체를 위한 slab_t 구조체의 크기와 슬랩이 관리할 obejct들의 포인터 역할을 할 kmem_bufctl_t 구조체들의 크기의 합이 된다. 즉 하나의 슬랩에는 슬랩 자신을 위한 데이터와 슬랩이 관리할 object들을 위한 데이터들이 들어간다는 뜻이다.

또 하드웨어 캐쉬의 경계를 맞추기 위해서 크기가 더 커질 수도 있다. 만약 슬랩의 크기가 40바이트였다면 64바이트로 조정해야한다.

```
        /*
         * If the slab has been placed off-slab, and we have enough space then
         * move it on-slab. This is at the expense of any extra colouring.
         */
        if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
                flags &= ~CFLGS_OFF_SLAB;
```

```
            left_over -= slab_size;
        }
```

만약 외부 슬랩을 사용하도록 플래그를 설정했는데 페이지에 남는 공간이 슬랩의 크기보다 크다면 굳이 외부 슬랩을 사용할 필요가 없어진다. 내부 슬랩이 더 효율이 좋기 때문이다. 따라서 다시 플래그를 조정해서 내부 슬랩을 사용하도록 한다.

```
        /* Offset must be a multiple of the alignment. */
        offset += (align-1);
        offset &= ~(align-1);
        if (!offset)
            offset = L1_CACHE_BYTES;
```

인자로 넘어온 offset을 하드웨어 캐쉬 경계에 맞춘다.

```
        cachep->colour_off = offset;
        cachep->colour = left_over/offset;
```

offset이 결정되었으므로 cache의 colour 속성을 설정해준다. (하드웨어 캐시에 관한 내용은 잘 모름)

```
        /* init remaining fields */
        if (!cachep->gfporder && !(flags & CFLGS_OFF_SLAB))
            flags |= CFLGS_OPTIMIZE;

        cachep->flags = flags;
        cachep->gfpflags = 0;
        if (flags & SLAB_CACHE_DMA)
            cachep->gfpflags |= GFP_DMA;
        spin_lock_init(&cachep->spinlock);
        cachep->objsize = size;
        INIT_LIST_HEAD(&cachep->slabs_full);
        INIT_LIST_HEAD(&cachep->slabs_partial);
        INIT_LIST_HEAD(&cachep->slabs_free);

        if (flags & CFLGS_OFF_SLAB)
            cachep->slabp_cache = kmem_find_general_cachep(slab_size,0);
        cachep->ctor = ctor;
        cachep->dtor = dtor;
```

```
        /* Copy name over so we don't have problems with unloaded modules */
        strcpy(cachep->name, name);
```

지금까지 계산한 cache의 속성들을 cache에 저장하고 나머지 속성들도 초기화한다.

외부 슬랩을 사용한다면 kmem_find_general_cachep() 함수를 사용해서 cache_sizes 에서 적당한 슬랩을 위한 메모리를 얻어온다.

```
        /* Need the semaphore to access the chain. */
        down(&cache_chain_sem);
        {
                struct list_head *p;

                list_for_each(p, &cache_chain) {
                        kmem_cache_t *pc = list_entry(p, kmem_cache_t, next);

                        /* The name field is constant - no lock needed. */
                        if (!strcmp(pc->name, name))
                                BUG();
                }
        }

        /* There is no reason to lock our new cache before we
         * link it in - no one knows about it yet...
         */
        list_add(&cachep->next, &cache_chain);
        up(&cache_chain_sem);
opps:
        return cachep;
}
```

주석에 나온 그대로 cache들의 리스트에 추가하고 cache의 포인터를 반환하고 함수를 끝낸다.

2. kmem_cache_alloc

linux 2.4.19

```
/**
```

```
 * kmem_cache_alloc - Allocate an object
 * @cachep: The cache to allocate from.
 * @flags: See kmalloc().
 *
 * Allocate an object from this cache.  The flags are only relevant
 * if the cache has no available objects.
 */
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
        return __kmem_cache_alloc(cachep, flags);
}


static inline void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
        unsigned long save_flags;
        void* objp;

        kmem_cache_alloc_head(cachep, flags);
```

kmem_cache_alloc_head() 함수는 할당에 필요한 기본적인 검사들을 하게된다.

```
try_again:
        local_irq_save(save_flags);
        objp = kmem_cache_alloc_one(cachep);
        local_irq_restore(save_flags);
        return objp;
```

kmem_cache_alloc_one 매크로는 slabs_partial이나 slabs_free에서 객체를 할당할 수 있으면 할당하고 실패하면 캐쉬에 새로운 슬랩을 추가해서 다시 할당을 시도한다.

```
alloc_new_slab:
        local_irq_restore(save_flags);
        if (kmem_cache_grow(cachep, flags))
                /* Someone may have stolen our objs.  Doesn't matter, we'll
                 * just come back here again.
                 */
                goto try_again;
```

객체 할당에 실패하면 슬랩을 늘리고 다시 시도한다.

```
        return NULL;
}


/*
 * Returns a ptr to an obj in the given cache.
 * caller must guarantee synchronization
 * #define for the goto optimization 8-)
 */
#define kmem_cache_alloc_one(cachep)                                    ₩
({                                                                      ₩
        struct list_head * slabs_partial, * entry;              ₩
        slab_t *slabp;                                                  ₩
                                                                        ₩
        slabs_partial = &(cachep)->slabs_partial;               ₩
        entry = slabs_partial->next;                                    ₩
```

- slabs_partial에 슬랩이 있는지 검사한다. 만약 slabs_partial에 슬랩이 있다면 이 슬랩에는 빈 객체가 있다는 것을 의미하므로 이 슬랩에서 객체를 얻어오면 된다.

```
        if (unlikely(entry == slabs_partial)) {                 ₩
                struct list_head * slabs_free;                  ₩
                slabs_free = &(cachep)->slabs_free;             ₩
                entry = slabs_free->next;                       ₩
```

slabs_partial에 슬랩이 없다면 slabs_free에 있는 슬랩에서 객체를 얻는다.

```
                if (unlikely(entry == slabs_free))              ₩
                        goto alloc_new_slab;                    ₩
```

slabs_free에도 슬랩이 없다면 현재 캐쉬에는 빈 객체가 전혀 없다는 것이다. 따라서 새로 슬랩을 추가하고 다시 객체 할당을 시도해야한다.

```
                list_del(entry);                                ₩
                list_add(entry, slabs_partial);                 ₩
```

만약 slabs_free에있는 슬랩에서 객체를 할당했다면 이 슬랩은 더 이상 빈 객체만을 가진 슬랩이 아니다. 따라서 slabs_partial 리스트로 옮겨야한다.

```
        }                                                       ₩
                                                                ₩
        slabp = list_entry(entry, slab_t, list);                ₩
        kmem_cache_alloc_one_tail(cachep, slabp);               ₩
```

객체를 얻은 슬랩의 포인터를 계산하고 kmem_cache_alloc_one_tail() 함수를 호출한다.

})

3. kmalloc/kfree

linux 2.6.11 기준

```
void *kmalloc(size_t size, int flags)
{
        struct cache_sizes *csizep = malloc_sizes;
        kmem_cache_t * cachep;

        for (; csizep->cs_size; csizep++) {
```

32부터 131,072바이트까지 미리 만들어놓은 general cache에서 요청된 사이즈보다 크고 가장 비슷한 크기의 캐시를 찾는다

```
                if (size > csizep->cs_size)
                        continue;
                if (flags & __GFP_DMA)
                        cachep = csizep->cs_dmacachep;
                else
                        cachep = csizep->cs_cachep;
```

DMA에 사용될 메모리 영역인지 확인 후 캐시 디스크립터를 찾고, 찾아진 캐시에서 객체를 얻는다.

```
                return kmem_cache_alloc(cachep, flags);
        }

        return NULL;
}

void kfree(const void *objp)
{
        kmem_cache_t *c;
        unsigned long flags;

        if (!objp)
                return;
```

```
        local_irq_save(flags);
        c = (kmem_cache_t *)(virt_to_page(objp)->lru.next);
```
페이지 디스크립터의 lru.next에는 캐시의 포인터가 저장되어 있다.
```
        kmem_cache_free(c, (void *)objp);
```
객체를 캐시에 반환한다.
```
        local_irq_restore(flags);
}
```

4. kmem_getpages/kmem_freepages

linux 2.6.17 기준

```
1450 static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
1451 {
1452     struct page *page;
1453     void *addr;
1454     int i;
1455
1456     flags |= cachep->gfpflags;
1457 #ifndef CONFIG_MMU
1458     /* nommu uses slab's for process anonymous memory allocations, so
1459      * requires __GFP_COMP to properly refcount higher order allocations"
1460      */
1461     page = alloc_pages_node(nodeid, (flags | __GFP_COMP), cachep->gfporder);
1462 #else
1463     page = alloc_pages_node(nodeid, flags, cachep->gfporder);
```
alloc_page_node() 함수는 버디 시스템에 있는 __alloc_pages() 함수를 호출해서 할당된 페이지의 디스크립터 포인터를 반환한다.

```
1464 #endif
1465     if (!page)
1466         return NULL;
1467     addr = page_address(page);
```
페이지 디스크립터에서 virtual 필드에 저장된 가상 주소를 반환한다.

```
1468
1469     i = (1 << cachep->gfporder);
```

i는 할당된 페이지 개수가 된다.

```
1470    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
1471        atomic_add(i, &slab_reclaim_pages);
```

slab_reclaim_pages는 시스템에 메모리가 부족할 때 슬랩에서 얼마나 메모리를 뺏어올 수 있는지를 나타내는 전역 변수이다. 몇몇 중요하지 않은 캐시는 시스템에 메모리가 부족할 때 우선적으로 메모리를 반환하도록 만들 수 있다.

```
1472    add_page_state(nr_slab, i);
```

전체 페이지의 상태 정보를 관리하는 struct page_state 라는 구조체가 있다. 이 구조체의 nr_slab 필드는 현재 슬랩에 사용된 페이지가 몇 개인지를 기억한다.

```
1473    while (i--) {
1474        __SetPageSlab(page);
```

페이지 디스크립터의 상태 정보에 PG_slab 플래그를 셋팅한다. PG_slab는 이 페이지가 현재 슬랩에 할당되어 있다는 표시이다.

```
1475        page++;
1476    }
1477    return addr;
```

커널 공간에서의 가상 주소가 반환된다. 즉 유저 메모리 공간의 주소가 아니라 0xC0000000 이상의 커널 공간 주소가 반환된다. (이유는 페이지 관리 테이블 초기 설정을 참고해야 함)

```
1478 }
```

```
1480 /*
1481  * Interface to system's page release.
1482  */
1483 static void kmem_freepages(struct kmem_cache *cachep, void *addr)
1484 {
1485    unsigned long i = (1 << cachep->gfporder);
1486    struct page *page = virt_to_page(addr);
1487    const unsigned long nr_freed = i;
1488
1489    while (i--) {
1490        BUG_ON(!PageSlab(page));
1491        __ClearPageSlab(page);
```

1492 page++;
1493 }
페이지 디스크립터에 PG_slab 플래그를 삭제한다.

1494 sub_page_state(nr_slab, nr_freed);
struct page_state에서 nr_slab 카운터에 해제될 페이지 개수를 뺀다.

1495 if (current->reclaim_state)
1496 current->reclaim_state->reclaimed_slab += nr_freed;
1497 free_pages((unsigned long)addr, cachep->gfporder);
버디 시스템의 free_pages() 함수를 호출한다.

1498 if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
1499 atomic_sub(1 << cachep->gfporder, &slab_reclaim_pages);
1500 }
slab_reclaim_pages 변수에서 반환된 페이지 개수를 뺀다.

5. __rmqueue

linux 2.6.17

562 /*
563 * Do the hard work of removing an element from the buddy allocator.
564 * Call me with the zone->lock already held.
565 */
566 static struct page *__rmqueue(struct zone *zone, unsigned int order)
memory zone을 찾은 상태에서 호출된다.

567 {
568 struct free_area * area;
569 unsigned int current_order;
570 struct page *page;
571
572 for (current_order = order; current_order < MAX_ORDER; ++current_order) {
요청된 order부터 MAX_ORDER까지 페이지 할당을 시도한다.

```
573        area = zone->free_area + current_order;
574        if (list_empty(&area->free_list))
575            continue;
576
577        page = list_entry(area->free_list.next, struct page, lru);
```
free_list.next는 struct page의 lru 필드와 연결되어 있다. container_of 매크로를 이용해서 lru 필드에서부터 page의 포인터를 찾아낸다.

```
578        list_del(&page->lru);
```
free_list에서 꺼낸다.

```
579        rmv_page_order(page);
```
페이지의 상태 플래그에서 PG_buddy 플래그를 지우고, private 필드를 초기화한다. (왜?)

```
580        area->nr_free--;
```
해당 free_area[]의 nr_free 필드가 감소한다.

```
581        zone->free_pages -= 1UL << order;
```
현재 memory zone에서 자유 페이지가

```
582        expand(zone, page, order, current_order, area);
```
만약 큰 페이지 블록을 쪼개서 할당했으면, 남은 페이지 블록을 해당 free_area에 연결한다.

```
583        return page;
584    }
585
586    return NULL;
587 }
```

6. __free_one_page

linux 2.6.17

```
334 static inline void __free_one_page(struct page *page,
335        struct zone *zone, unsigned int order)
336 {
```

337 unsigned long page_idx;

페이지의 번호

338 int order_size = 1 << order;

페이지 개수

339

340 if (unlikely(PageCompound(page)))

341 destroy_compound_page(page, order);

342

343 page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);

344

345 BUG_ON(page_idx & (order_size - 1));

346 BUG_ON(bad_range(zone, page));

347

348 zone->free_pages += order_size;

349 while (order < MAX_ORDER-1) {

루프를 돌 때 마다 점점 더 큰 페이지 블록으로 합쳐진다.

350 unsigned long combined_idx;

351 struct free_area *area;

352 struct page *buddy;

353

354 buddy = __page_find_buddy(page, page_idx, order);

현재 해제되는 페이지와 버디가 될 수 있는 페이지 포인터는 page에 해제할 페이지 개수를 더하면 된다.

355 if (!page_is_buddy(buddy, order))

356 break; /* Move the buddy up one level. */

버디로 추정되는 페이지의 디스크립터의 플래그를 읽어서 PG_buddy 플래그를 확인한다. 그리고 private 필드에 order와 같은 값이 저장되어 있으면 버디를 찾은 것이다.

357

358 list_del(&buddy->lru);

359 area = zone->free_area + order;

360 area->nr_free--;

361 rmv_page_order(buddy);

찾은 버디를 free_area[] 리스트에서 꺼낸다.

```
362         combined_idx = __find_combined_index(page_idx, order);
363         page = page + (combined_idx - page_idx);
364         page_idx = combined_idx;
365         order++;
```

새로 order가 높아진 페이지 블록을 얻었으므로, 새 페이지 블록에서 첫번째 페이지의 디스크립터 포인터를 계산한다. (정말 깔끔하고 감동적인 코드입니다!! ㅜ,.ㅜ)

```
366     }
367     set_page_order(page, order);
```

최종적으로 얻어진 페이지 블록 중에서 첫번째 페이지 디스크립터를 설정한다. 상태 플래그에 PG_buddy를 셋팅하고 private 필드에 order를 저장한다.

```
368     list_add(&page->lru, &zone->free_area[order].free_list);
369     zone->free_area[order].nr_free++;
370 }
```

참고 자료

- Memory Management in Linux - Desktop Companion to the Linux Source Code
- Understanding the Linux Kernel
- Functional Callgraph of the Linux VM
- Process Management in Linux – Desktop Companion to the Linux Source Code