# 리눅스 메모리 관리 알고리즘 분석

## 버디 / 슬랩 알고리즘 소스 분석 및 알고리즘 해설

**2004.9.7**

김기오 **(gurugio@gmail_nospam.com)**

**http://www.asmlove.co.kr**

알림.

이 문서는 개인적인 참고 자료의 모음과 연구실 내부에서 신입생 교육에 사용될 개인적인 자료입니다. 혹시 저작권이나 기타 위반 사항 등이 발견되면 언제라도 연락 주시면 즉시 바로잡도록 하겠습니다. 이 문서는 개인 적인 참고 용도로만 사용될 수 있습니다.

김기오 **(gurugio@gmail_nospam.com.nospam)**

# 문서 개요

범용 커널의 설계에서 메모리 관리는 씨피유와 물리 메모리 디바이스, 시스템 버스같은 하드웨어의 설계뿐만 아니라 커널의 사용 목적과 이용되는 환경, 대상으로 하는 유저등 많은 사항들을 고려해야한다. 특히 커널 메모리 할당기의 설계는 시스템의 성능과 안정성에 절대적인 영향을 미친다. 커널 메모리 할당기가 곧 범용 메모리 할당의 설계에도 이어지므로 메모리 관리에 대한 설계의 가장 큰 밑바탕이 된다고 할 수 있다.

메모리 관리는 매우 중요하며 민감한 주제이므로 초창기 커널의 최초 적합, 최적 적합과 같은 자원 맵 할당부터 현대 Solaris 2.4와 Linux 2.6에서 사용되는 slab allocator까지 많은 메모리 할당 알고리즘이 개발되어왔고 지금도 활발히 연구되고 있다. 페이징 기법의 사용으로 인해 획기적인 단편화 줄임이 이루어졌고 대용량 메모리의 관리가 편리해졌고 가상 메모리와 특히 요구 페이징으로 인해 현대적인 서버 시스템으로서 필요한 대용량 메모리 처리가 가능해졌다.

시스템이 부팅할 때 커널은 우선 자기 자신을 위한 메모리를 할당해야한다. 커널 자신의 코드와 정적 데이터 구조등과 미리 정의된 메모리 풀등을 예약해야한다. 그 후 페이지 수준 할당기가 남은 물리 메모리를 관리하기 시작하게 된다. 이 페이지 수준 할당기는 전체 물리 메모리의 양과 사용중인 메모리 공간과 자유 공간을 관리해야하고 전체적인 메모리 부족 현상에 대비해야하며 갑작스러운 메모리 감소 현상을 막을 수 있어야 한다. 이런 페이지 수준 할당기가 효율적이고 안정적이어야 그 위에서 작은 크기의 메모리 할당이 제대로 이루어질 수 있게 되는 것이다.

메모리 할당기를 평가하는 중요한 기준은 얼마나 메모리 낭비를 최소화할 수 있는가이다. 물리 메모리는 당연히 한정되어 있으므로 할당기는 메모리 공간을 효율적으로 이용해야한다. 이 효율성을 평가하는 기준으로 활용 계수가 있다. 활용 계수는 메모리 요청을 처리하기 위해 필요한 메모리의 양과 요청된 메모리의 양의 비율을 말한다. 이상적으로는 100%의 활용도가 요구되지만 실제로는 50%정도 수준까지 인정된다고 알려져있다.

메모리 낭비의 가장 큰 원인은 단편화 현상이다. 시스템에 남아있는 전체적인 자유 메모리는 요청되는 메모리의 양보다 크지만 자유 메모리들이 조각으로 나눠져있어서 할당해줄 수 없는 상태를 말한다. 할당기는 인접한 자유 메모리들을 합쳐서 큰 덩어리를 만들어서 단편화를 줄여야 한다.

커널 메모리 할당기는 최대한 빨라야하는데 커널 메모리 할당기의 성능이 인터럽트 핸들러등 커널 전반적인 시스템의 성능에 큰 영향을 미치기 때문이다. 평균 지연과 최악 지연 시간이 모두 민감하게 작용하는데 만약 Real-Time 특성을 필요로 하는 상황이라면 최악 지연 시간이 시스템에 더욱 큰 영향을 미칠 것이다.

메모리 할당기는 여러 다양한 메모리 요청에 맞게 단순한 프로그래밍 인터페이스를 가져야 한다. 대표적인 방법은 이미 널리 사용되고 있는 malloc(), free()와 같은 표준 라이브러리와 유사한 인터페이스를 가지는 것이다.

```
void *malloc(size);
void free(address)
```

이 방법의 이점은 메모리 해제시에 해제할 메모리의 크기를 알 필요가 없다는 것이다. 대부분의 커널 함수는 한 개의 메모리 덩어리를 할당해서 다른 서브 시스템으로 복사해주는데 이 때 이 메모리들의 크기에 대해 이미 알고있다면 해제 작업을 단순화 시킬 수 있다.

또 다른 요구사항은 할당된 페이지를 늘리거나 줄일 수 있는 유연성이다. 메모리를 할당받은 후 일부만 해제하거나 할당받은 메모리를 늘릴 수 있다면 더욱 유용할 것이다.

또 메모리 할당기는 주기적인 메모리 요청이나 돌발적인 메모리 요청 모두에 적합해야한다. 일정 기간에 주기적으로 동작하는 데이터베이스 처리등과 같은 작업을 위한 대비책과 돌발적인 인터럽트나 네트워크 패킷 처리등과 같은 메모리 요청에도 대비하고 있어야한다. 많은 시스템에서 이를 위해 미리 다양한 크기의 메모리 버퍼들을 준비해두어서 요청때마다 반환해주는 방식을 사용한다.

메모리 할당기와 페이징 시스템의 유기적인 동작도 필요하다. 사용되지 않는 메모리들을 스왑하고 스왑 공간에서 다시 메모리로 불러들이거나 메모리 부족 현상을 이기기위해 메모리를 확보하는 등의 많은 작업을 필요로 한다.

이번 논문에서는 현재 사용되고 있는 대표적인 커널인 Linux의 2.4.19버전 소스를 이용하여 대표적인 페이지 단위 메모리 할당기인 버디 시스템과 커널 메모리 할당기인 슬랩 할당자를 분석하고 이해할 것이다. 이 두 알고리즘은 이미 여러 번 그 우수성이 증명되었고 널리 사용되고 있으므로 따로 성능을 평가하거나 시뮬레이션 작업을 하지는 않고 그 동작 원리를 이해하는데 중점을 둘 것이다.

# PART (I) SLAB ALLOCATOR

## 1 슬랩 할당자 개론

### 1.1 페이지 기반 시스템의 메모리 할당시 발생하는 문제

실제적으로 커널에 필요한 메모리들은 대부분 작으면서 자주 사용되는 데이터 구조들을 저장하기 위한 메모리들이다. 그런데 물리 메모리를 다루는 단위는 페이지가 된다. 페이지의 크기보다 작은 메모리를 요청했을 때 페이지 전체를 할당해준다면 메모리의 낭비가 생기게 된다. 슬랩 할당자 이전에 개발된 알고리즘들에서 이런 내부 단편화 비율이 20%에서 40%까지 나타난다는 결과 보고(Uresh Vahalia. UNIX Internals: The New Frontiers)가 있다. 따라서 물리적인 메모리에 독립적이면서 페이지 단위보다 작은 메모리 크기의 요청을 처리할 할당 기법이 필요하다.

결과적으로 말한다면 슬랩 할당자에서의 오버 헤드는 슬랩과 캐쉬 디스크립터 자체를 위한 메모리와 하드웨어 캐쉬에 맞추기 위한 정렬, 페이지에 객체들을 할당시키고 남은 객체의 크기보다 작은 공간이 되어 내부 단편화를 획기적으로 줄일 수 있게된다.

### 1.2 슬랩 할당자의 특성

- 내부 단편화를 줄인다. 이를 위해 슬랩 할당자는 페이지 내부에 일정한 크기의 작은 메모리 버퍼들을 할당해서 가지고있다. 만약 커널이 특정한 크기의 버퍼를 요청하면 이 메모리를 반환해주게 된다.
- mm_struct 처럼 일반적으로 사용되는 객체를 위해 메모리 풀을 만들어서 제공한다. 이것은 객체들을 위한 메모리를 할당하거나 복잡한 객체들을 생성하고 파괴하는 오버헤드를 줄이기 위해서이다.
- 최대한 하드웨어 캐쉬를 활용하기위해 slab coloring과 같은 몇 가지 기술을 사용한다.
- 결론적으로 커널에서 메모리 요청이 있을 때마다 메모리 할당을 위한 과정을 처음부터 시작하는 것이 아니라 미리 준비해놓은 동일한 크기의 메모리를 반환해줌으로서 성능과 효율을 동시에 올릴 수 있게 해준다는 것이 가장 큰 특징이 된다.

### 1.3 terminology

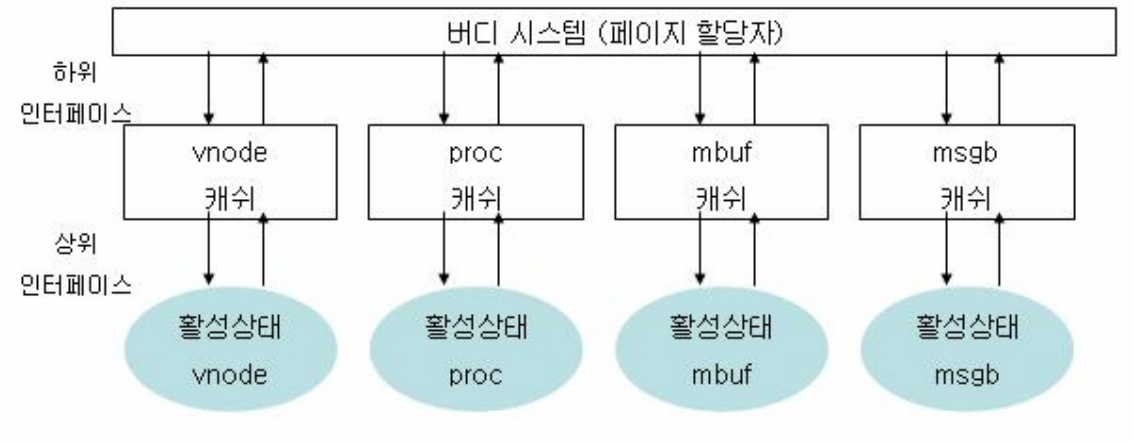
- cache
  최근에 사용된 객체들 중에 같은 타입의 객체들끼리 모아놓은 것이다. 슬랩 할당자에서 가장 상위의 개념이된다. dentry_cache같은 이름을 가진다.
- slab
  객체들이 저장되는 단위가 되며 하나이상의 페이지 프레임으로 이뤄진다. 하나의 cache는 여러 개의 슬랩으로 이뤄진다.
- object
  가장 작은 단위이다. 슬랩에 저장되며 실제적으로 메모리 요청에 대응해서 할당되는 메모리 단위이다.

## 1.4 슬랩 할당자의 설계와 인터페이스

리눅스에서 사용되는 슬랩 할당자에 대한 알고리즘은 Bonwick의 논문(Jeff Bonwick. The Slab Allocator: An Object Caching Kernel Memory Allocator)을 따른다.

슬랩 할당자는 객체 캐쉬들의 집합체로 구성된다. 각 캐쉬는 단일 타입의 객체들을 포함한다. 보통 커널은 원하는 타입의 캐쉬에서 객체를 받아쓰고 다시 캐쉬로 반환한다. 또 할당자는 캐쉬에 할당된 메모리를 늘려서 캐쉬에 속해있는 객체의 수를 늘리거나 어디에도 사용되지 않는 완전히 비어있는 메모리의 크기를 유지하기 위해 캐쉬의 메모리를 줄이기도한다.

슬랩의 하위 인터페이스는 버디 시스템과 통신한다. 캐쉬를 유지하기 위한 메모리를 버디 시스템에 요청하고 필요없는 메모리를 버디 시스템에 반납하기도 한다. 상위 인터페이스는 메모리를 요청하는 클라이언트들과 통신하게된다.



[ 그림 1-1 ]

슬랩은 비어있는 슬랩과 일부만 사용중인 슬랩, 모두 사용된 슬랩으로 나눠진다. 최초로 cache가 생기면 빈 슬랩을 할당받고 슬랩을 조금씩 사용하다가 모두 사용하면 다시 빈 슬랩을 할당받는다. 따라서 하나의 cache에는 세가지 종류의 슬랩이 모두 존재할 수 있다.

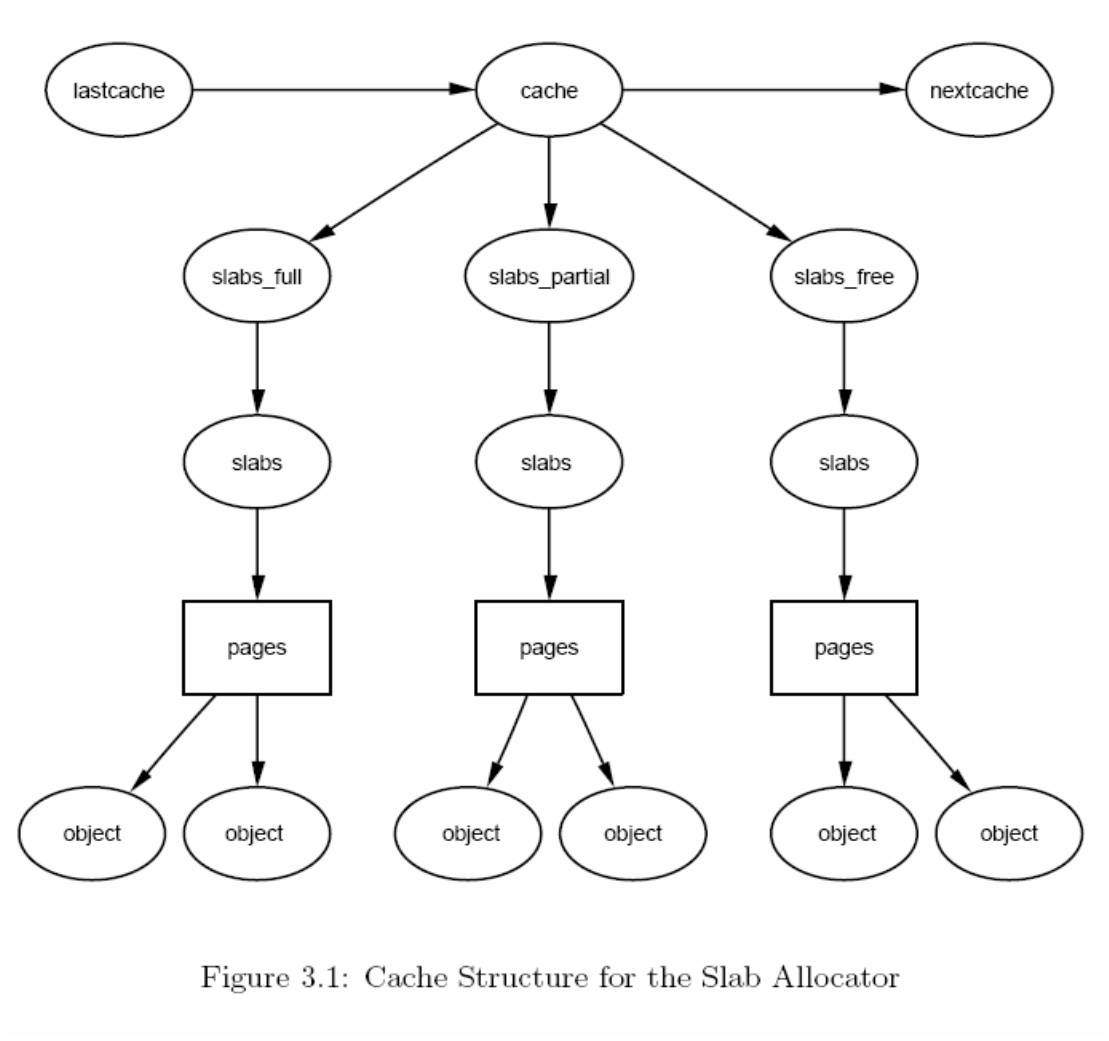



Figure 3.1: Cache Structure for the Slab Allocator

[ 그림 1-2 ]

## 1.5 구현

### 1.5.1 캐쉬 관리

커널은 특정 타입의 객체를 관리하기 위해, kmem_cache_create() 함수로 캐쉬를 초기화한다. 다음과 같이 사용한다.

```
cachep = kmem_cache_create(name, size, align, ctor, dtor);
```

name은 객체의 이름을 나타내는 문자열이고 size는 객체의 바이트 단위 크기, align은 객체들을 정렬할 크기를 말한다. ctor과 dtor은 객체의 생성자와 소멸자를 말하는데 리눅스에서는 사용하지 않으므로 NULL로 지정한다.

캐쉬로부터 객체를 할당하는 함수는 kmem_cache_alloc()이다. 객체를 해제하는 함수는 kmem_cache_free()이다. 각각 다음과 같이 사용된다.

```
objp = kmem_cache_alloc(cachep, flags);
kmem_cache_free(cachep, objp);
```

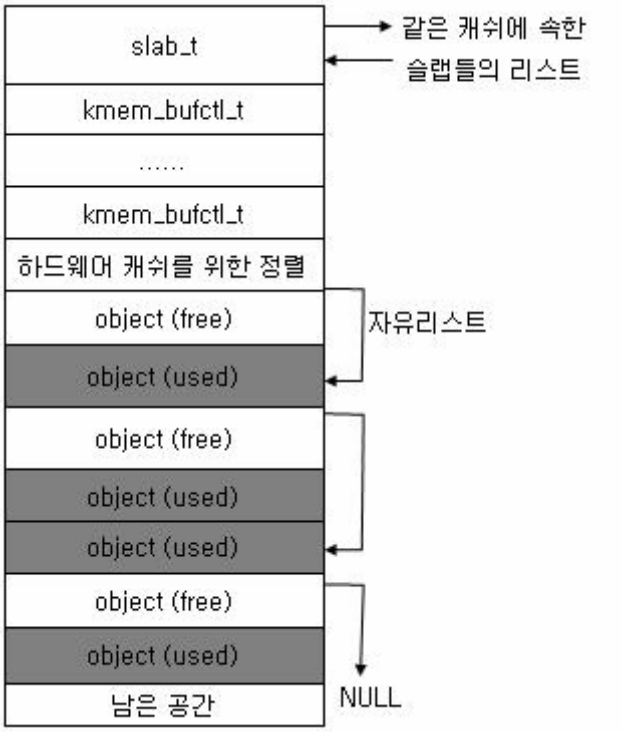
캐쉬가 사용중인 객체만 가지고있어서 더 이상 객체를 할당해줄 수 없으면 kmem_cache_grow() 함수를 호출하여 버디 시스템에서 페이지를 얻고 객체들을 생성한다. 이 함수는 페이지를 얻어서 슬랩을 설정하고 페이지를 객체들을 위한 메모리로 나눈다.

만약 시스템에 메모리가 모자라거나 추가적으로 빈 페이지를 만들어야할 때 kmem_cache_reap() 함수로 캐쉬에 있는 사용중이지 않은 슬랩을 해제하고 슬랩에 할당되었던 페이지들을 비워서 버디 시스템으로 넘긴다.

### 1.5.2 슬랩/객체 관리

슬랩 할당자는 객체의 크기에 따라서 큰 객체와 작은 객체로 나눠서 다른 구조의 슬랩을 사용한다.

먼저 작은 객체들을 관리하는 슬랩에 대해서 알아보면 보통 한 슬랩에 한 페이지를 할당해서 사용하게 된다. 하나의 슬랩은 슬랩 자체의 디스크립터인 slab_t 와 객체들을 관리하기 위한 포인터(kmem_bufctl_t)와 객체들의 저장 공간, 정렬이나 객체를 할당하고 마지막으로 남겨진 부분처럼 사용되지 않는 공간으로 나눠진다.




[ 그림 1-3 ]

남은 공간이란 객체들을 할당하고 페이지의 끝에 남은 객체의 크기보다 작은 공간을 말한다. 예를 들어 inode 객체가 300바이트이고 한 페이지(4096 바이트)에 객체들을 할당한다면 13개의 객체를 만들고 104바이트가 남겨지게된다. 즉 내부 단편화 비율이 약 2%가 된다. 커널이 사용하는 대부분의 객체들의 크기는 이처럼 수백 바이트정도로 작으므로 내부 단편화 현상을 획기적으로 줄일 수 있게된다.

kmem_bufctl_t 데이터는 비어있는 객체들의 위치 정보를 가지고 있고 하드웨어 캐쉬를 위한 정렬은 첫번째 객체의 오프셋을 하드웨어 캐쉬에 맞게 맞춰주는 것이다. 이는 슬랩 할당자보다는 하드웨어 캐쉬에 대한 내용이 중심이므로 다루지 않겠다.

큰 객체를 다루는 슬랩에서는 위에서처럼 페이지 내에 모든 데이터들을 저장하지 않는다. 슬랩 자체를 위한 slab_t와 kmem_bufctl_t들은 이런 경우에 사용되도록 미리 준비된 별도의 전용 공간에 저장하고 페이지에는 객체들만을 저장하게된다. 예를 들어 객체의 크기가 1024바이트라면 페이지에는 4개의 객체를 저장하고 별도의 공간에 기타 데이터들을 저장하는 것이 더욱 효율적일 것이다.

## 1.6 about the '/proc/slabinfo'

시스템에 있는 모든 cache와 슬랩에 대한 정보를 보기 위해서는 /proc/slabinfo를 열어보면 된다. 파일에는 다음과 같은 필드들이 있다.

| | |
|---|---|
| cache-name | vm_area_struct와 같이 문자열로된 cache의 이름 |
| num-active-objs | 현재 사용중인 객체들의 수 |
| total-objs | 총 객체의 수 |
| obj-size | 객체의 바이트 단위 크기 |
| num-active-slabs | 사용중인 객체를 하나라도 가지고있는 슬랩의 수 |
| total-slabs | 총 슬랩의 수 |
| num-pages-per-slab | 슬랩 한개를 만들기위해 필요한 페이지 수 |

# 2 슬랩 할당자에 사용되는 데이터 분석

## 2.1 매크로

```
/* Macros for storing/retrieving the cachep and or slab from the
 * global 'mem_map'. These are used to find the slab an obj belongs to.
 * With kfree(), these are used to find the cache which an obj belongs to.
 */
#define        SET_PAGE_CACHE(pg,x)    ((pg)->list.next = (struct list_head *)(x))
#define        GET_PAGE_CACHE(pg)      ((kmem_cache_t *)(pg)->list.next)
#define        SET_PAGE_SLAB(pg,x)     ((pg)->list.prev = (struct list_head *)(x))
#define        GET_PAGE_SLAB(pg)       ((slab_t *)(pg)->list.prev)
```

page 디스크립터에는 버디 알고리즘에서 사용하는 리스트 포인터가 있다. 이 포인터는 슬랩 알고리즘에서는 필요하지 않으므로 해당 페이지가 속하는 슬랩과 캐쉬의 포인터를 저장하는데 사용한다. list.next는 페이지가 속한 캐쉬의 포인터를 가지게되고 list.prev는 슬랩의 포인터를 저장하게된다.

## 2.2 상수

```
/*
 * Parameters for kmem_cache_reap
 */
```

```
#define REAP_SCANLEN        10
#define REAP_PERFECT        10
```

kmem_cache_reap() 함수에서 사용된다.

```
#define cache_chain (cache_cache.next)
```

cache_cache는 캐쉬들을 가지고있는 캐쉬이다. 여기서 cache_cache.next는 사용되고있는 캐쉬중에서 첫번째 캐쉬를 가르킨다. 따라서 cache_chain은 사용되고있는 캐쉬들의 리스트의 헤더가 된다.

```
/*
 * Absolute limit for the gfp order
 */
#define         MAX_GFP_ORDER       5       /* 32 pages */

 /*
 * Do not go above this order unless 0 objects fit into the slab.
 */
#define         BREAK_GFP_ORDER_HI      2
#define         BREAK_GFP_ORDER_LO      1
static int slab_break_gfp_order = BREAK_GFP_ORDER_LO;
```

kmem_cache_create() 함수에서 사용된다.

### 2.3 구조체

```
struct kmem_cache_s {
/* 1) each alloc & free */
        /* full, partial first, then free */
        struct list_head slabs_full;
        struct list_head slabs_partial;
        struct list_head slabs_free;
        unsigned int            objsize; /* size of the objects */
        unsigned int            flags;  /* constant flags */
        unsigned int            num;    /* # of objs per slab */
```

```
/* 2) slab additions /removals */
        /* order of pgs per slab (2^n) */
        unsigned int            gfporder;

        /* force GFP flags, e.g. GFP_DMA */
        unsigned int            gfpflags;

        size_t                  colour;         /* cache colouring range */
        unsigned int            colour_off;     /* colour offset */
        unsigned int            colour_next;    /* cache colouring */
        kmem_cache_t            *slabp_cache;
        unsigned int            growing;

/* 3) cache creation/removal */
        char                    name[CACHE_NAMELEN];
        struct list_head next;
};
```

캐쉬의 디스크립터이다.

- name 캐쉬의 이름
- slabs_full 여유 객체가 없는 슬랩 디스크립터의 원형 이중 연결 리스트
- slabs_partial 여유 객체와 사용 중인 객체를 포함하는 슬랩 디스크립터의 원형 이중 연결 리스트
- slabs_free 여유 객체만 포함하는 슬랩 디스크립터의 원형 이중 연결 리스트
- spinlock 멀티프로세서 시스템에서 캐쉬를 동시에 접근하지 않도록 보호하는 스핀락
- num 슬랩 하나에 들어있는 객체의 수 (같은 캐쉬의 슬랩들은 크기가 같다)
- objsize 캐쉬에 들어 있는 객체의 크기
- gfporder 슬랩 하나에 들어가는 연속된 페이지 프레임 수에 로그를 취한 값
- ctor, dtor 객체의 생성자와 소멸자 메소드. 리눅스에서는 사용하지 않음
- next 캐쉬 디스크립터의 이중 연결 리스트
- flags 캐쉬의 정적인 속성들을 나타내는 플래그

- dflags 캐쉬의 동적인 속성들을 나타내는 플래그. kmem_cache_grow() 에서는 해당 캐쉬가 현재 새로운 슬랩들을 추가하고 있는지를 기록한다.
- gfpflags 페이지 프레임을 할당할 때 버디 시스템으로 전달하는 플래그

```
typedef struct slab_s {
        struct list_head list;
        unsigned long           colouroff;
        /* points to the first object (either allocated or free) inside the slab */
        void                    *s_mem;                 /* including colour offset */
        unsigned int            inuse;          /* num of objs active in slab */
        /* points to the first free object (if any) in the slab */
        kmem_bufctl_t           free;
} slab_t;
```

슬랩의 디스크립터이다.

- inuse 슬랩에서 현재 할당한 객체의 수
- s_mem 슬랩에 있는 첫번째 객체를 가르킨다. 사용중이거나 아니거나 구별하지 않고 첫번째 객체를 말한다.
- free 슬랩에서 사용하고있지 않는 객체중 첫번째를 가르킨다.
- list 캐쉬에서 slabs_full, slabs_partial, slabs_free 세가지 이중 연결 리스트를 관리하는데 이 리스트에 있는 슬랩들 간의 이중 연결 리스트이다.

슬랩의 디스크립터는 두가지 종류가 있다.

- 외부 슬랩 디스크립터 : 슬랩 외부에 cache_sizes가 가르키는 일반 캐쉬 중 하나에 저장한다.
- 내부 슬랩 디스크립터 : 슬랩 내부에 슬랩에 할당된 첫번째 페이지 프레임의 시작 부분에 저장한다.

슬랩 할당자는 객체의 크기가 512바이트보다 작거나 내부 단편화에 의해 슬랩 내에 슬랩 디스크립터와 뒤에서 설명할 객체 디스크립터를 담을 충분한 공간이 있으면 후자의 방법을 선택한다.

```
/* Size description struct for general caches. */
typedef struct cache_sizes {
        size_t              cs_size;
        kmem_cache_t *cs_cachep;
```

```
        kmem_cache_t *cs_dmacachep;
} cache_sizes_t;
```

특정 크기의 캐쉬를 저장하기위해 미리 할당된 메모리 공간을 관리한다.
32, 64, 128, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 130172까지 크기의 메모리 버퍼들을 가지고 있으면서 특정 크기의 메모리 요청이 있을 때 반환해 준다.

## 2.4 상수

```
/* Max number of objs-per-slab for caches which use off-slab slabs.
 * Needed to avoid a possible looping condition in kmem_cache_grow().
 */
static unsigned long offslab_limit;
```

외부 슬랩 (off-slab) 에서 하나의 slab이 처리할 수 있는 최대한의 object의 수. 만약 슬랩을 32바이트의 cache_sizes에서 메모리를 얻어서 저장했다고 하면 슬랩의 크기가 20바이트라고 가정했을 때 12바이트가 남는다. 여기에 object를 관리하기 위한 kmem_bufctl_t가 4바이트이므로 3개의 object를 관리할 수 있다. 즉 offslab_limit가 3이 되는 것이다. 주석에 나온대로 슬랩을 추가하거나 cache를 생성할 때 사용되는 변수이다.

```
static cache_sizes_t cache_sizes[] = {
        {     32,        NULL, NULL},
        {     64,        NULL, NULL},
        {    128,        NULL, NULL},
        {    256,        NULL, NULL},
        {    512,        NULL, NULL},
        {   1024,        NULL, NULL},
        {   2048,        NULL, NULL},
        {   4096,        NULL, NULL},
        {   8192,        NULL, NULL},
        {  16384,        NULL, NULL},
        {  32768,        NULL, NULL},
        {  65536,        NULL, NULL},
        { 131072,        NULL, NULL},
```

```
        {       0,          NULL, NULL}
};
```

미리 일정한 크기의 메모리들을 할당해서 사용한다. 슬랩을 위한 메모리를 여기에서 할당한다.

```
/* internal cache of cache description objs */
static kmem_cache_t cache_cache = {
        slabs_full:     LIST_HEAD_INIT(cache_cache.slabs_full),
        slabs_partial:  LIST_HEAD_INIT(cache_cache.slabs_partial),
        slabs_free:     LIST_HEAD_INIT(cache_cache.slabs_free),
        objsize:sizeof(kmem_cache_t),
        flags:          SLAB_NO_REAP,
        spinlock:       SPIN_LOCK_UNLOCKED,
        colour_off:     L1_CACHE_BYTES,
        name:           "kmem_cache",
};
```

주석에 적힌대로 캐쉬들을 위한 캐쉬이다. 새로운 캐쉬를 만들 때마다 디스크립터를 저장하기위한 메모리를 cache_cache가 가진 슬랩들에서 할당받는다.

# 3 소스 분석

## 3.1 caches관련 함수

캐쉬는 struct kmem_cache_s(kmem_cache_t) 데이터 형으로 관리된다. 몇 가지 중요한 항목을 설명하면 다음과 같다.

- 리스트 관련 항목
  struct list_head slabs_full 모든 객체가 사용중인 슬랩들의 리스트
  struct list_head slabs_partial 일부 객체만 사용중인 슬랩들의 리스트
  struct list_head slabs_free 사용중인 객체가 없는 슬랩들의 리스트
  struct list_head next 캐쉬들의 리스트에서 다음 캐쉬를 가르킨다.
- 객체의 특성
  char name[CACHE_NAMELEN] 캐쉬의 이름
  unsigned int objsize 객체의 크기

unsigned int flags 객체의 저장을 위한 속성을 지정하는 플래그들
unsigned int num 슬랩 하나에 저장되는 객체의 개수

■ 객체 생성
void (*ctor)() 객체의 생성자
void (*dtor)() 객체의 소멸자

캐쉬는 하드웨어 캐쉬의 성능을 끌어올리기 위해 서로 다른 슬랩에 있는 객체들을 다른 오프셋에 저장한다. 이 오프셋은 슬랩에 객체를 할당하고 남는 공간의 크기에 따라 결정되는데 캐쉬가 SLAB_HWCACHE_ALIGN 속성을 가지면 L1 하드웨어 캐쉬에 맞게 정렬하기위해 L1_CACHE_BYTES의 배수로 오프셋을 정하고 그렇지 않으면 BYTES_PER_WORD의 배수로 오프셋을 정하게된다. (kmem_cache_estimate() 함수 참조)
캐쉬를 처음 생성할 때 하나의 슬랩에 몇 개의 객체가 저장될 수 있는지 또 몇 바이트가 남게 되는지를 계산한다. 계산 결과에 따라 캐쉬 디스크립터에 있는 다음 두 항목이 결정된다.

colour 사용될 수 있는 오프셋의 개수
colour_off 객체 오프셋의 크기

객체들의 오프셋을 정함으로서 하드웨어 캐쉬에 서로 중첩되지 않도록 하게된다.
예를 들어 설명하면 만약 첫번째 객체의 시작 주소가 0이고 객체를 저장하고 남는 공간의 크기가 100바이트, 정렬할 크기가 펜티엄 2의 L1 하드웨어 캐쉬의 크기인 32바이트라고 가정해보자. 그러면 첫번째 슬랩의 객체들의 오프셋은 0이 될 것이고 두번째 슬랩에서는 32, 세번째는 64, 네번째는 96, 다섯번째는 100보다 오프셋이 클 수 없으므로 다시 0으로 설정될 것이다. 즉 colour의 값은 4(지정할 수 있는 오프셋이 0, 32, 64, 96이므로 4개)이 되고 colour_off는 32가 되는 것이다.
이렇게 각 슬랩들의 객체들은 같은 하드웨어 캐쉬 라인에 놓이지 않게된다.

### 3.1.1 kmem_cache_init()

general cache를 위한 초기화를 한다. cache_cache는 cache들을 위한 cache이다. 즉 새로운 cache를 만들려면 cache를 위한 메모리를 할당받아야하는데 이를 위해서는 cache를 나타내는 kmem_cacahe_t 데이터도 미리 slab으로 만들어져있어서 메모리에 할당되어있어야한다. 다시 말해서 각 cache들을 저장할 메모리를 미리 할당해놓고있어야 한다는 것이다.이 함수가 호출하는 kmem_cache_estimate() 함수를 보면 페이지의 order를 0으로 설정하는 것을 볼 수 있다. 즉 페이지를 한개만 잡고 거기에 kmem_cache_t 데이터를 슬랩처럼 만들어놓고 이 cache들을 관리하는 cache가 cache_cache가 되는 것이다.

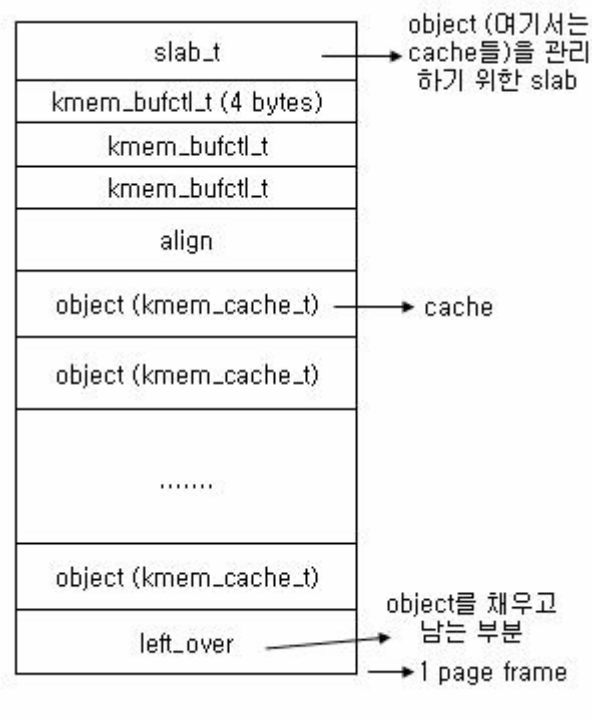
결론적으로 slab을 관리하는 cache들도 slab처럼 cache의 cache에게 관리되는 것이다.

우선 cache_cache 구조체를 초기화한다.

```
/* internal cache of cache description objs */
static kmem_cache_t cache_cache = {
        slabs_full:     LIST_HEAD_INIT(cache_cache.slabs_full),
        slabs_partial:  LIST_HEAD_INIT(cache_cache.slabs_partial),
        slabs_free:     LIST_HEAD_INIT(cache_cache.slabs_free),
        objsize:sizeof(kmem_cache_t),
        flags:          SLAB_NO_REAP,
        spinlock:       SPIN_LOCK_UNLOCKED,
        colour_off:     L1_CACHE_BYTES,
        name:           "kmem_cache",
};
```

- objsize가 kmem_cache_t 구조체의 크기가 된다. 즉 cache를 object로 관리하는 cache의 cache가 된다는 것을 말한다. flags에 SLAB_NO_REAP으로 설정하는 것은 절대 cache_cache를 위해 할당된 페이지가 해제되서는 않되기 때문이다.
- 이름을 “kmem_cache”로 설정한다. /proc/slabinfo에 보면 가장 처음으로 나오는 cache가 이것이다. 첫번째 컬럼이 이름이고 두번째 컬럼이 현재 사용중인 object의 갯수가 되는데 kmem_cache의 object는 cache를 나타내므로 만약 이 값이 66으로 되어있으면 시스템에 cache가 66개라는 뜻이다. kmem_cache를 제외한 cache의 수를 세보면 이 값과 일치하는 것을 알 수 있을 것이다.

다음은 cache_cache에 할당된 페이지의 그림이다. 각 object들이 cache이 되고 object들을 관리하기 위한 slab이 페이지 첫 부분에 있다. 각 object의 리스트를 관리하기 위한 kmem_bufctl_t이 slab의 바로 다음에 위치하고있고 align은 하드웨어 캐쉬를 위해 정렬을 하기위해 공간을 비워둔 것이다. 페이지 마지막에는 object들을 할당하고 남은 공간이 있는데 당연하게 이 공간의 크기는 object의 크기보다 작다.




[ 그림 3-1 ]

```
/* Initialisation - setup the `cache' cache. */
void __init kmem_cache_init(void)
{
        size_t left_over;
```

- 페이지에서 cache나 cache_cache를 할당하고 남은 메모리

```
        INIT_LIST_HEAD(&cache_chain);
```

- cache들의 리스트를 설정한다.

```
        kmem_cache_estimate(0, cache_cache.objsize, 0,
                        &left_over, &cache_cache.num);
```

- 함수에 대한 설명은 다음 장에서 한다.
- left_over를 계산하고 1개의 페이지안에 몇개의 kmem_cache_t 데이터가 들어갈 수 있는지 cache_cache.num에 저장해준다.

```
        cache_cache.colour = left_over/cache_cache.colour_off;
        cache_cache.colour_next = 0;
```

- slab colouring에 대해서는 하드웨어 캐쉬에대한 내용이므로 future work로 남긴다.

}

### 3.1.2 kmem_cache_sizes_init()

start_kernel() 함수안에서 호출된다. 32바이트에서 128K 까지 크기의 cache를 초기화한다. /proc/slabinfo에 보면 size-32라고 나와있는 cache들이다. cache_sizes 테이블을 보면 다음과 같다.

```
/* Size description struct for general caches. */
typedef struct cache_sizes {
        size_t             cs_size;
        kmem_cache_t *cs_cachep;
        kmem_cache_t *cs_dmacachep;
} cache_sizes_t;

static cache_sizes_t cache_sizes[] = {
        {      32,          NULL, NULL},
        .....
```

cs_size는 cache가 관리할 object의 크기를 말하고 cs_cachep는 cache에 대한 포인터이다. 이전에 설명한 kmem_cache_init() 함수에서 만들어도 되지만 이 cache들을 미리 정해져있으므로 이렇게 테이블로 고정시켜놓고 사용하면 더 효율적이다.

kmem_cache_sizes_init() 함수는 kmem_cache_create() 함수를 사용해서 cache를 생성하고 이 테이블의 cachep 포인터에 포인터를 저장한다.

```
/* Initialisation - setup remaining internal and general caches.
 * Called after the gfp() functions have been enabled, and before smp_init().
 */
void __init kmem_cache_sizes_init(void)
{
        cache_sizes_t *sizes = cache_sizes;
```

- cache_sizes 테이블안에 cache들을 처리한다.

```
char name[20];

do {
        /* For performance, all the general caches are L1 aligned.
         * This should be particularly beneficial on SMP boxes, as it
         * eliminates "false sharing".
         * Note for systems short on memory removing the alignment will
         * allow tighter packing of the smaller caches. */
        snprintf(name, sizeof(name), "size-%Zd",sizes->cs_size);
```

- name 버퍼에 처리할 cache의 이름을 저장한다. cache의 이름은 /proc/slabinfo에 나와있는데로 size->object_size 가 되는데 object의 크기는 cache_sizes 구조체의 cs_size 항목에 저장되어 있으므로 그대로 문자열로 바꿔서 사용하면 된다.

```
        if (!(sizes->cs_cachep =
                kmem_cache_create(name, sizes->cs_size,
                                0, SLAB_HWCACHE_ALIGN, NULL, NULL)))
{
                BUG();
        }
```

- cache를 생성해서 cache_sizes 구조체의 cachep 항목에 저장한다. cache를 생성하는 kmem_cache_create() 함수는 다음 장에 설명한다. 이 함수는 cache의 이름과 object의 크기를 인자로 받아서 알맞은 cache를 생성해서 그 포인터를 반환해준다.

```
        sizes++;
    } while (sizes->cs_size);
```

- 다음 크기의 cache를 처리한다. 32바이트부터 128k의 cache를 생성한다.

### 3.1.3 kmem_cache_create()

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 2^3=8배, 즉 32K보다 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 확인해 보고 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 슬랩이 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 슬랩이 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 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.

```
/* 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.

### 3.1.4 kmem_cache_estimate()

cache를 만들면서 얼마나 많은 object들이 한 슬랩에 저장될 수 있고 남는 공간은 얼마인지 알아내야한다. 이 일을 하는 함수가 kmem_cache_estimate()이다.

```
/* Cal the num objs, wastage, and bytes left over for a given slab size. */
static void kmem_cache_estimate (unsigned long gfporder, size_t size,
                 int flags, size_t *left_over, unsigned int *num)
```

- gfporder : 각 슬랩에 할당될 페이지 수의 order
- size : object의 크기

- flags : cache의 플래그
- left_over : 슬랩에 사용되지 않고 남겨질 바이트 수
- num : 슬랩에 들어갈 수 있는 object의 갯수
- left_over와 num은 결국 이 함수가 계산해서 반환할 값이다.

```
{
        int i;
        size_t wastage = PAGE_SIZE<<gfporder;
        size_t extra = 0;
        size_t base = 0;
```

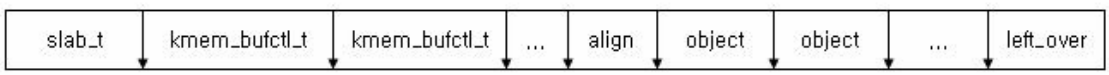
- wastage는 함수 내부에서 계속 줄어들면서 슬랩안에 사용되지 않고 남겨질 바이트 수를 계산하는데 사용된다. 그래서 슬랩에 사용된 전체 메모리 크기로 초기화된다.

```
        if (!(flags & CFLGS_OFF_SLAB)) {
                base = sizeof(slab_t);
                extra = sizeof(kmem_bufctl_t);
        }
```

- 내부 슬랩을 사용할 때는 페이지 내부에 슬랩을 위한 slab_t와 슬랩이 object들을 관리하기 위해 필요한 kmem_bufctl_t 데이터가 들어간다. 이 데이터들 다음에 object가 저장되므로 이 데이터들의 크기를 기억한다.
- 외부 슬랩을 사용할 때는 이 데이터들이 페이지 밖에 저장되므로 건너뛴다.

```
        i = 0;
        while (i*size + L1_CACHE_ALIGN(base+i*extra) <= wastage)
                i++;
```

- i는 슬랩이 관리할 수 있는 object의 갯수가 된다. i*size는 object들에 할당될 메모리의 크기이고 base+i*extra는 slab_t와 kmem_bufctl_t 가 차지할 크기이다. 슬랩안에는




[ 그림 3-2 ]

이런 모습으로 데이터가 들어가는데 object들을 하드웨어 캐쉬에 정렬 시키기 위해서는 slab_t부터 align까지가 하드웨어 캐쉬에 정렬되야만한다. 이

값들의 합이 슬랩에 할당된 메모리 크기 wastage보다 작거나 같아야 한다는 것이다.

```
if (i > 0)
        i--;
```

- 위의 while 루프에서 계산 값이 wastage보다 클 때까지 i를 증가시키므로 1을 감소시켜 주어야 한다.

```
if (i > SLAB_LIMIT)
        i = SLAB_LIMIT;
```

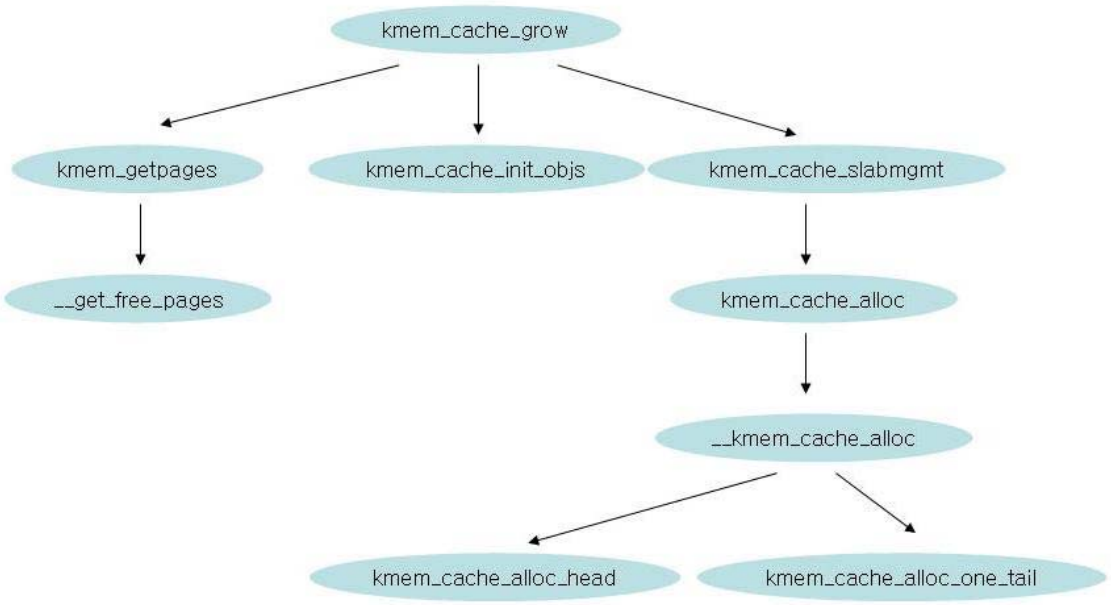
- SLAB_LIMIT은 하나의 슬랩안에 저장할 수 있는 최대한의 object의 갯수이다.

```
*num = i;
wastage -= i*size;
wastage -= L1_CACHE_ALIGN(base+i*extra);
*left_over = wastage;
}
```

- num 은 슬랩이 가지고있는 object의 수이다.
- wastage 에서 object들이 차지한 메모리 크기 (i*size) 를 빼낸다.
- wastage 에서 slab_t 와 kmem_bufctl_t 들이 차지한 메모리 크기를 뺀다.
- wastage 는 결국 슬랩에서 남은 공간의 크기가 된다.

### 3.1.5 kmem_cache_grow()

kmem_cache_create()는 cache를 생성하지만 아무 slab도 가지고 있지 않다. 따라서 slab_full, slab_partial, slab_free 리스트는 모두 비어있다. slab_partial에 비어있는 object가 없거나 slab_free에 아무 slab도 없으면 cache를 증가시키기위해 사용되는 함수가 kmem_cache_grow()이다.




[ 그림 3-3 ]

개략적으로 kmem_cache_grow()이 하는 일은 다음과 같다

- 기본적인 에러 체크
- 슬랩안의 object를 위한 colour offset 계산
- 슬랩에 사용된 메모리 할당과 슬랩 디스크립터 작성
- 슬랩이 사용할 페이지와 cache 디스크립터의 링크 작성
- 슬랩안의 object들을 초기화
- cache에 슬랩 추가

```
/*
 * Grow (by 1) the number of slabs within a cache.  This is called by
 * kmem_cache_alloc() when there are no active objs left in a cache.
 */
static int kmem_cache_grow (kmem_cache_t * cachep, int flags)
{
        slab_t  *slabp;
        struct page     *page;
        void            *objp;
        size_t           offset;
        unsigned int     i, local_flags;
```

```
unsigned long    ctor_flags;
unsigned long    save_flags;


/* Get colour for the slab, and cal the next value. */
offset = cachep->colour_next;
cachep->colour_next++;
if (cachep->colour_next >= cachep->colour)
        cachep->colour_next = 0;
offset *= cachep->colour_off;
cachep->dflags |= DFLGS_GROWN;
```

- colour_next 은 현재 cache에 사용될 offset을 계산하기 위해 kmem_cache_create에서 계산된다. colour은 cache에 모두 몇 개의 다른 colour이 있는지 저장하고 있으므로 colour_next가 colour보다 커질 수 없다.
- 현재 사용될 object의 offset은 colour_next * colour_off 가 된다.
- 다음에 생성될 슬랩을 위해 colour_next를 증가시킨다. (colour_next+1) % colour 이 된다.

```
cachep->dflags |= DFLGS_GROWN;
cachep->growing++;
```

- cache에 새로운 슬랩이 추가되었다는 것을 기록한다. 현재 슬랩은 모두 비어있는 상태이므로 만약 시스템의 메모리 확보를 위해 kswapd 이 활성화되면 이 슬랩은 해제될 수 있다. 이것을 막기 위해 플래그를 설정해둔다.

```
/* A series of memory allocations for a new slab.
 * Neither the cache-chain semaphore, or cache-lock, are
 * held, but the incrementing c_growing prevents this
 * cache from being reaped or shrunk.
 * Note: The cache could be selected in for reaping in
 * kmem_cache_reap(), but when the final test is made the
 * growing value will be seen.
 */

/* Get mem for the objs. */
if (!(objp = kmem_getpages(cachep, flags)))
```

```
        goto failed;
```

- __alloc_pages() 함수를 호출해서 페이지 프레임을 얻는다.

```
/* Get slab management. */
if (!(slabp = kmem_cache_slabmgmt(cachep, objp, offset, local_flags)))
        goto opps1;
```

- local_flags 를 검사해서 cache가 내부 슬랩을 사용하는지 외부 슬랩을 사용하는지 알아낸다. 그리고 외부 슬랩이면 위의 그림에서 본 듯이 kmem_cache_alloc 함수를 사용해서 슬랩에 사용할 메모리를 따로 얻고 내부 슬랩을 사용한다면 kmem_getpages() 함수에서 얻은 페이지 프레임 안에 슬랩을 저장한다.

```
/* Nasty!!!!!! I hope this is OK. */
i = 1 << cachep->gfporder;
page = virt_to_page(objp);
do {
        SET_PAGE_CACHE(page, cachep);
        SET_PAGE_SLAB(page, slabp);
        PageSetSlab(page);
        page++;
} while (--i);
```

- SET_PAGE_CACHE 매크로는 페이지의 next 포인터에 cache의 주소를 저장한다.
- SET_PAGE_SLAB 매크로는 페이지의 prev 포인터에 슬랩의 주소를 저장한다.
- 페이지의 next, prev 포인터는 버디 알고리즘에서 페이지 프레임 관리를 위해 사용하므로 현재는 사용되지 않는다. 그래서 이 포인터들을 사용해도 버디 알고리즘과는 아무런 문제가 없다.
- PageSetSlab 매크로는 페이지의 플래그에 PG_slab 비트를 설정한다.
- 루프를 돌면서 슬랩에 할당된 모든 페이지에 같은 처리를 해준다.

```
kmem_cache_init_objs(cachep, slabp, ctor_flags);
```

- cache에 object를 초기화하기 위한 constructor가 있으면 실행해서 각 object들을 초기화한다.

```
cachep->growing--;
```

- 나중에 메모리 확보를 위해 슬랩을 해제할 수 있도록 한다.

```
/* Make slab active. */
list_add_tail(&slabp->list, &cachep->slabs_free);
```

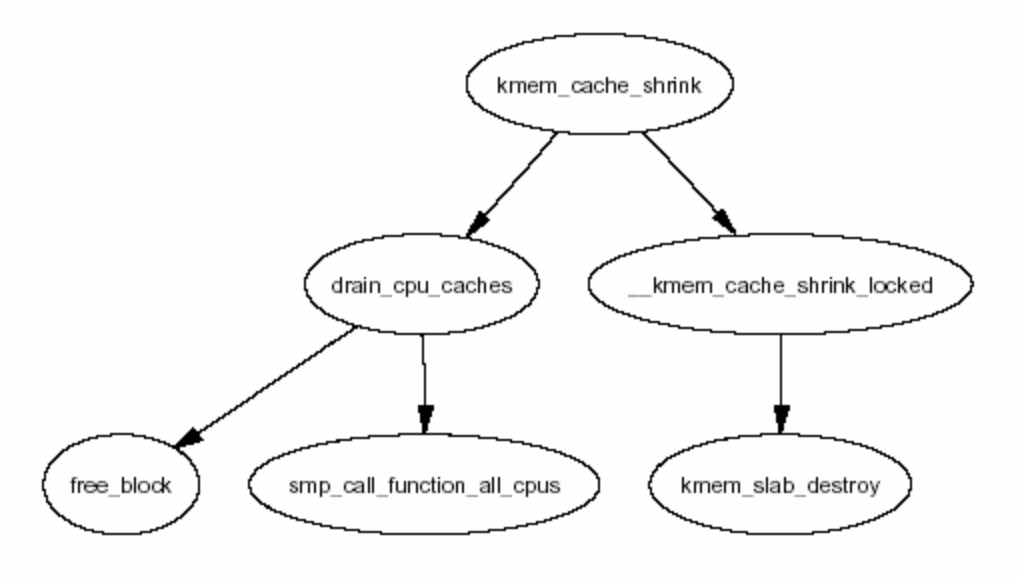
- 슬랩을 cache의 비어있는 슬랩 리스트에 추가한다.

```
return 1;
```

- 종료한다.

### 3.1.6 kmem_cache_shrink()

시스템에 메모리가 부족해졌을 때 kswapd 데몬이 깨어나 메모리 확보를 위해 각 cache의 slabs_free 리스트를 검사한다. 이 때 리스트에 사용되지 않는 슬랩이 있으면 그 슬랩을 해제하고 슬랩에 할당된 페이지 프레임을 버디 알고리즘의 빈 페이지 리스트로 반환시킨다.




[ 그림 3-4 ]

```
/**
 * kmem_cache_shrink - Shrink a cache.
 * @cachep: The cache to shrink.
 *
```

```
 * Releases as many slabs as possible for a cache.
 * Returns number of pages released.
 */
int kmem_cache_shrink(kmem_cache_t *cachep)
{
        int ret;

        if (!cachep || in_interrupt() || !is_chained_kmem_cache(cachep))
                BUG();

        drain_cpu_caches(cachep);

        spin_lock_irq(&cachep->spinlock);
        ret = __kmem_cache_shrink_locked(cachep);
        spin_unlock_irq(&cachep->spinlock);

        return ret << cachep->gfporder;
}
```

- 디버깅과 에러 체크를 빼면 __kmem_cache_shrink_locked() 함수를 호출하는 일이 전부이다.
- drain_cpu_caches() 함수는 SMP 에서 사용하므로 언급하지 않는다.
- __kmem_cache_shrink_locked() 함수는 없앤 슬랩의 개수를 반환하므로 하나의 슬랩에 몇 페이지가 할당되는지를 가지고 계산하면 몇 개의 페이지 프레임을 반환했는지 알 수 있다.

#### 3.1.7 __kmem_cache_shrink_locked()

이 함수는 인자로 받은 cache의 slabs_free 리스트에 있는 모든 슬랩마다 kmem_slab_destroy() 함수를 호출해서 슬랩을 해지시킨다.

```
/*
 * Called with the &cachep->spinlock held, returns number of slabs released
 */
static int __kmem_cache_shrink_locked(kmem_cache_t *cachep)
{
```

```
        slab_t *slabp;
        int ret = 0;

        /* If the cache is growing, stop shrinking. */
        while (!cachep->growing) {
```

- 만약 cache가 kmem_cache_grow() 함수로 슬랩을 생성하고있는 도중이라면 함수를 종료한다.

```
                struct list_head *p;

                p = cachep->slabs_free.prev;
                if (p == &cachep->slabs_free)
                        break;
```

- slabs_free 리스트에 슬랩이 없어질 때까지 반복해서 실행한다.

```
                slabp = list_entry(cachep->slabs_free.prev, slab_t, list);
                list_del(&slabp->list);
```

- slabs_free 리스트에서 슬랩을 빼낸다.

```
                kmem_slab_destroy(cachep, slabp);
                ret++;
```

- 슬랩을 없애고 없앤 슬랩의 개수를 기록한다.

```
        }
        return ret;
}
```

- 없어진 슬랩의 개수를 반환한다.

### 3.1.8 __kmem_slab_destroy()

슬랩에 들어있는 각 object들을 돌면서 destructor 를 호출해서 object들을 해제한다.

```
/* Destroy all the objs in a slab, and release the mem back to the system.
 * Before calling the slab must have been unlinked from the cache.
```

```
 * The cache-lock is not held/needed.
 */
static void kmem_slab_destroy (kmem_cache_t *cachep, slab_t *slabp)
{
        if (cachep->dtor
        ) {
                int i;
                for (i = 0; i < cachep->num; i++) {
                        void* objp = slabp->s_mem+cachep->objsize*i;
                        if (cachep->dtor)
                                (cachep->dtor)(objp, cachep, 0);
                }
        }
```

- 모든 object들을 돌면서 destructor를 호출한다.

```
        kmem_freepages(cachep, slabp->s_mem-slabp->colouroff);
```

- kmem_freepages()함수는 free_pages() 함수를 호출해서 버디 시스템에 빈 페이지를 넘긴다.

```
        if (OFF_SLAB(cachep))
                kmem_cache_free(cachep->slabp_cache, slabp);
}
```

- 만약 외부 슬랩을 사용한다면 슬랩에 사용된 메모리도 해제하고 함수를 종료한다.

### 3.1.9 kmem_cache_destroy()

cache를 제거하는 일은 해당 cache를 사용하는 모듈이 커널에서 제거되었을 때에만 호출된다. 모듈이 여러 번 커널에 등록되었다가 삭제되는 경우에도 cache가 중복되지 않도록 하기위해서 사용된다.

cache를 없애는 과정은 다음과 같다.

① cache chain에서 cache를 지운다.

② 모든 슬랩을 없앤다.

③ cache_cache 에서 cache의 디스크립터를 지운다.

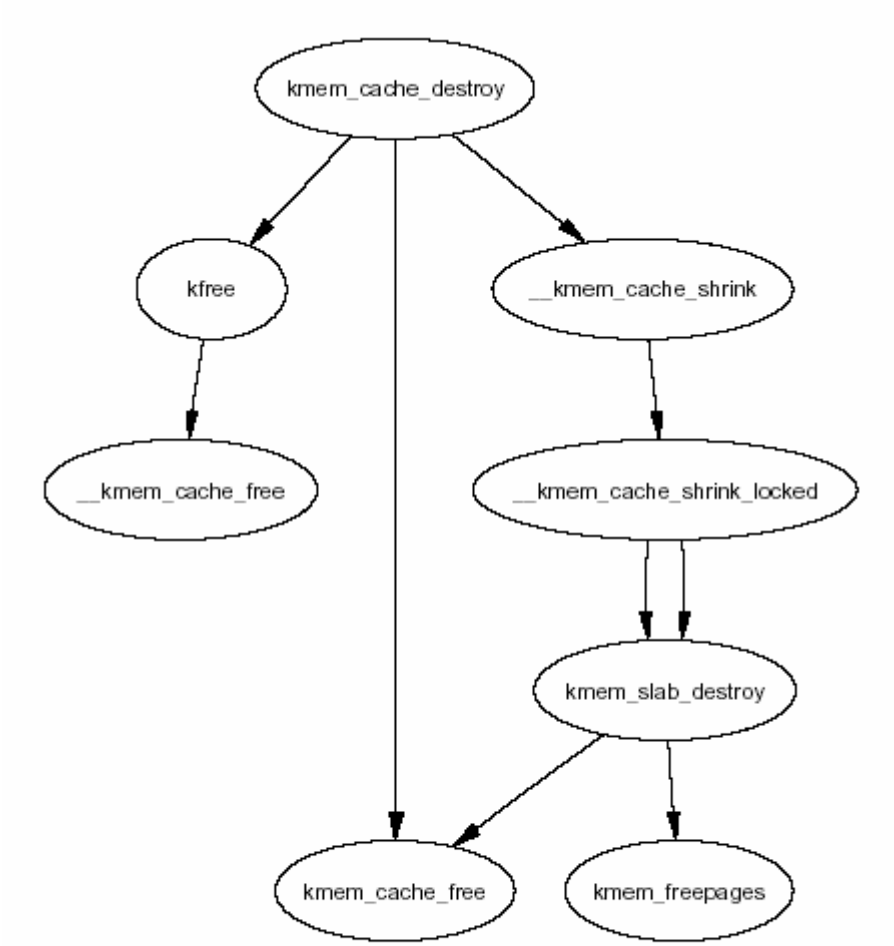



Figure 3.5: kmem_cache_destroy

[ 그림 3-5 ]

```
/**
 * kmem_cache_destroy - delete a cache
 * @cachep: the cache to destroy
 *
 * Remove a kmem_cache_t object from the slab cache.
 * Returns 0 on success.
 *
 * It is expected this function will be called by a module when it is
 * unloaded.  This will remove the cache completely, and avoid a duplicate
 * cache being allocated each time a module is loaded and unloaded, if the
 * module doesn't have persistent in-kernel storage across loads and unloads.
```

```
 *
 * The caller must guarantee that noone will allocate memory from the cache
 * during the kmem_cache_destroy().
 */
int kmem_cache_destroy (kmem_cache_t * cachep)
{
        if (!cachep || in_interrupt() || cachep->growing)
                BUG();
```

- sanity check

```
        /* Find the cache in the chain of caches. */
        down(&cache_chain_sem);
        /* the chain is never empty, cache_cache is never destroyed */
        if (clock_searchp == cachep)
                clock_searchp = list_entry(cachep->next.next,
                                                kmem_cache_t, next);
        list_del(&cachep->next);
        up(&cache_chain_sem);
```

- cache chain을 다루기위한 세마포어를 얻고 해당 cache를 찾아서 리스트에서 지운다.

```
        if (__kmem_cache_shrink(cachep)) {
                printk(KERN_ERR "kmem_cache_destroy: Can't free all
objects %p₩n",
                        cachep);
                down(&cache_chain_sem);
                list_add(&cachep->next,&cache_chain);
                up(&cache_chain_sem);
                return 1;
        }
```

- cache에 있는 빈 슬랩들을 모두 해제한다. 만약 그래도 cache에 슬랩이 남아있다면 __kmem_cache_shrink() 함수는 true를 반환하고 cache를 제거하는 일은 중단된다.
- 따라서 다시 cache chain에 넣고 함수를 종료한다.

```
        kmem_cache_free(&cache_cache, cachep);
```

```
        return 0;
}
```

- cache_cache에서 cache의 디스크립터를 없애고 함수를 종료한다.

#### 3.1.10 mem_cache_reap()

여유 메모리가 부족하게되면 kswapd 데몬이 빈 페이지를 만들기 시작한다. 이때 첫번째로 하는 일이 슬랩 할당자에게 cache의 크기를 줄이도록 하는 것이다. 크기를 줄일 cache를 찾는 일도 매우 긴 작업이다. 만약 시스템에 많은 cache가 있다면 REAP_SCANLEN 상수의 개수만큼의 cache만 검사해서 고르게된다. 마지막으로 검사된 cache는 clock_searchp 변수에 저장되어 같은 cache를 반복해서 검사하지 않도록 한다.

cache를 검사할 때마다 다음과 같은 작업을 한다.

① SLAB_NO_REAP 플래그가 설정되어있으면 넘어간다.

② cache가 늘어나고 있는 중이면, 즉 kmem_cache_growing() 함수의 중간과정에 있는 cache이면 넘어간다. kmem_cache_growing() 함수를 참고한다.

③ cache가 최근에 늘어난 상태, 즉 dflags에 DFLAGS_GROWN 플래그가 설정되어있으면 넘어간다. 하지만 플래그를 지워서 다음 검색에서는 크기를 줄일 수 있도록 한다.

④ slab_free 리스트를 검사해서 몇 개의 빈 페이지를 만들 수 있는지 계산해서 pages 라는 변수에 저장한다.

이 함수는 세 부분으로 구성되는데 첫번째는 일반적인 에러 검사와 변수들이고 두번째는 크기가 줄어들 cache를 선택하는 부분, 세번째 부분은 슬랩들을 해제하는 부분이다.

```
/**
 * kmem_cache_reap - Reclaim memory from caches.
 * @gfp_mask: the type of memory required.
 *
 * Called from do_try_to_free_pages() and __alloc_pages()
 */
int kmem_cache_reap (int gfp_mask)
{
```

```
slab_t *slabp;
kmem_cache_t *searchp;
kmem_cache_t *best_cachep;
unsigned int best_pages;
unsigned int best_len;
unsigned int scan;
int ret = 0;

if (gfp_mask & __GFP_WAIT)
        down(&cache_chain_sem);
else
        if (down_trylock(&cache_chain_sem))
                return 0;
```

- 만약 __GFP_WAIT 플래그가 설정되어있으면 이 함수를 호출한 함수가 언제든지 sleep 상태가 될 수 있다는 의미이다. 따라서 미리 cache 의 리스트에 접근할 세마포어를 얻어놓는다. 그렇지 않다면 세마포어를 얻을 수 있는지 시도하고 실패하면  종료한다.

```
scan = REAP_SCANLEN;
best_len = 0;
best_pages = 0;
best_cachep = NULL;
searchp = clock_searchp;
```

- REAP_SCANLEN은 검색할 cache의 갯수를 나타내는 상수다. 10으로 설정되어있다. searchp 변수는 지난번 검색에서 마지막으로 검사한 cache에 대한 포인터이다.

```
do {
```

- REAP_SCANLEN 갯수의 cache를 검사하면서 슬랩을 줄일 cache를 찾는다.

```
        unsigned int pages;
        struct list_head* p;
        unsigned int full_free;

        /* It's safe to test this without holding the cache-lock. */
        if (searchp->flags & SLAB_NO_REAP)
```

```
                goto next;
```

- reap을 하지 못하도록 설정된 cache이면 건너뛴다.

```
        spin_lock_irq(&searchp->spinlock);
        if (searchp->growing)
                goto next_unlock;
        if (searchp->dflags & DFLGS_GROWN) {
                searchp->dflags &= ~DFLGS_GROWN;
                goto next_unlock;
        }
```

- 현재 슬랩을 늘리고 있거나 늘린 직후라면 건너뛴다.

```
        full_free = 0;
        p = searchp->slabs_free.next;
        while (p != &searchp->slabs_free) {
                slabp = list_entry(p, slab_t, list);
#if DEBUG
                if (slabp->inuse)
                        BUG();
#endif
                full_free++;
                p = p->next;
        }
```

- slab_free 리스트에 몇 개의 슬랩이 들어있는지 계산해서 full_free 변수에 저장한다.

```
        /*
         * Try to avoid slabs with constructors and/or
         * more than one page per slab (as it can be difficult
         * to get high orders from gfp()).
         */
        pages = full_free * (1<<searchp->gfporder);
```

- 해제될 페이지의 갯수를 계산한다.

```
        if (searchp->ctor)
                pages = (pages*4+1)/5;
```

- 만약 생성자가 있다면 해제될 페이지의 갯수를 1/5로 줄인다.

```
if (searchp->gfporder)
        pages = (pages*4+1)/5;
```

- 만약 슬랩이 여러개의 페이지로 구성된다면 해제될 페이지의 갯수를 1/5로 줄인다.

```
if (pages > best_pages) {
        best_cachep = searchp;
        best_len = full_free;
        best_pages = pages;
        if (pages >= REAP_PERFECT) {
                clock_searchp = list_entry(searchp->next.next,
                                        kmem_cache_t,next);
                goto perfect;
        }
}
```

- 만약 지금까지 찾아낸 cache들보다 더 많은 페이지를 해제할 수있다면 현재의 cache에 대한 포인터를 best_cachep에 저장한다.
- REAP_PERFECT는 상수로 10을 가진다. 즉 10페이지 이상 해제할 수 있으면 현재 cache를 가지고 계속 처리하고 다른 cache를 검색하지는 않는다. 또 다음에 검색할 때 검색이 시작될 cache의 포인터를 clock_searchp에 저장해서 검색이 중복되지 않도록 한다.

```
next_unlock:
        spin_unlock_irq(&searchp->spinlock);
```

- cache의 슬랩이 늘어나고 있거나 늘어난 직후라면 이곳으로 점프한다. spinlock를 잠근 상태이므로 먼저 해제하고 다음 cache로 넘어가게된다.

```
next:
        searchp = list_entry(searchp->next.next,kmem_cache_t,next);
```

- 슬랩을 줄일 수 없도록 플래그가 설정되어있으면 이것으로 점프한다.

```
} while (--scan && searchp != clock_searchp);
```

- cache_chain에 있는 모든 cache를 검사하게 되거나 REAP_SCANLEN에 정해진 수만큼의 cache를 검사할 때까지 반복한다.

```
        clock_searchp = searchp;
```
- 다음 검색에서 검사를 시작할 cache의 포인터를 저장해둔다.

```
        if (!best_cachep)
                /* couldn't find anything to reap */
                goto out;
```
- 아직도 적당한 cache를 찾지 못했으면 함수를 종료한다.

```
        spin_lock_irq(&best_cachep->spinlock);
perfect:
        /* free only 50% of the free slabs */
        best_len = (best_len + 1)/2;
```
- slabs_free에서 해제할 수 있는 슬랩에서 절반만 해제한다.

```
        for (scan = 0; scan < best_len; scan++) {
                struct list_head *p;

                if (best_cachep->growing)
                        break;
                p = best_cachep->slabs_free.prev;
```
- slabs_free 리스트에서 슬랩들의 포인터를 얻는다.

```
                if (p == &best_cachep->slabs_free)
                        break;
```
- 리스트에 슬랩이 남아있지 않으면 루프를 멈춘다.

```
                slabp = list_entry(p,slab_t,list);
#if DEBUG
                if (slabp->inuse)
                        BUG();
#endif
                list_del(&slabp->list);
```
- 리스트에서 슬랩을 빼낸다.

```
                STATS_INC_REAPED(best_cachep);
```

- cache에 대한 정보에 통계를 위한 데이터들을 추가할 수 있다. 만약 이 통계를 위한 데이터가 있다면 이 데이터를 갱신해준다.

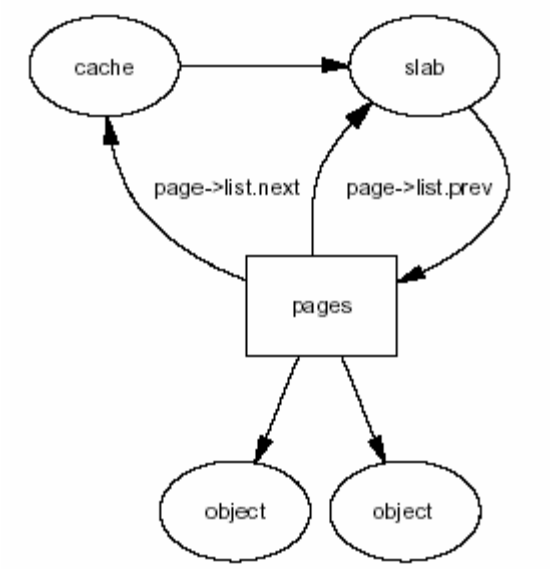
```
                /* Safe to drop the lock. The slab is no longer linked to the
                 * cache.
                 */
                spin_unlock_irq(&best_cachep->spinlock);
                kmem_slab_destroy(best_cachep, slabp);
                spin_lock_irq(&best_cachep->spinlock);
        }
        spin_unlock_irq(&best_cachep->spinlock);
        ret = scan * (1 << best_cachep->gfporder);
out:
        up(&cache_chain_sem);
        return ret;
}
```

- 슬랩을 파괴하고 슬랩을 제거하면서 생겨난 빈 페이지들의 갯수를 반환한다.

### 3.2 slabs관련 함수

첫 장에서 설명했듯이 슬랩은 하나 이상의 페이지로 구성되며 객체들을 관리하고 할당하기 위해 사용된다. 슬랩을 나타내는 구조체 slab_t 는 cache_sizes 테이블에 미리 할당된 메모리에 저장될 수도 있다. 이런 경우를 외부 슬랩(off slab) 이라고 부르며 슬랩 자체에 할당된 페이지에 객체들과 함께 저장되는 경우에는 내부 슬랩이라고 부른다. cache_sizes 테이블은 2의 배수의 크기를 가지는 메모리 블록을 저장하고있는 캐쉬를 말한다.

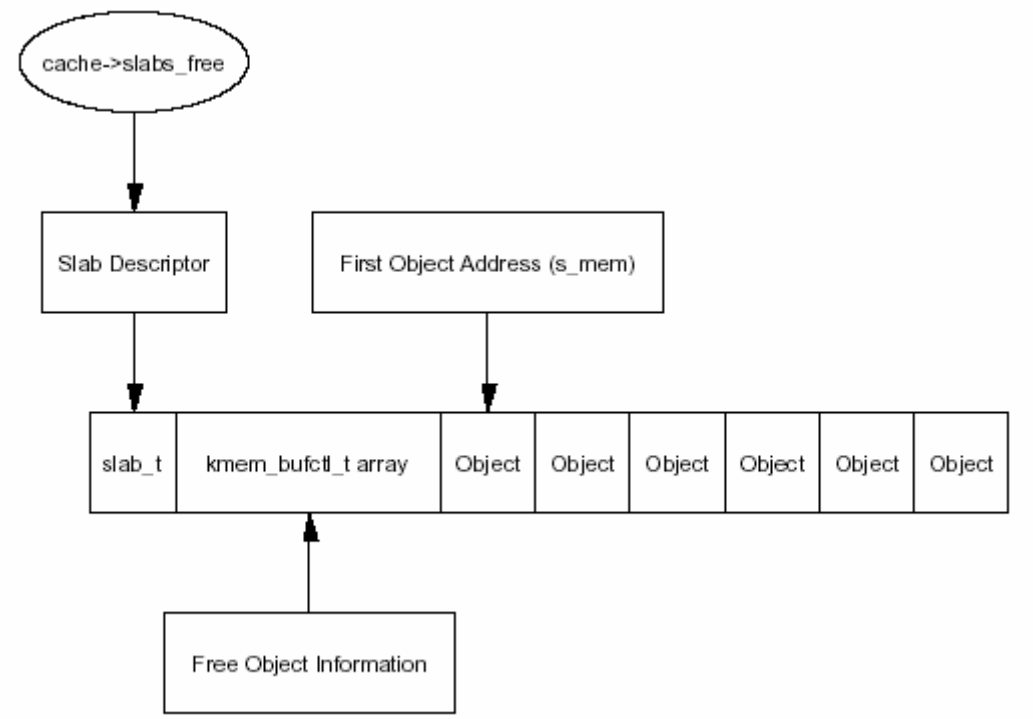
슬랩이나 객체들이 자신들이 속한 캐쉬나 슬랩에 대해서 알기위해서는 페이지의 디스크립터를 이용해야한다. pages->list.next 항목에 캐쉬의 포인터를 저장하고 pages->list.prev 항목에 슬랩의 포인터를 저장한다. 그림으로 나타내면 다음과 같다.




[ 그림 3-6 ]

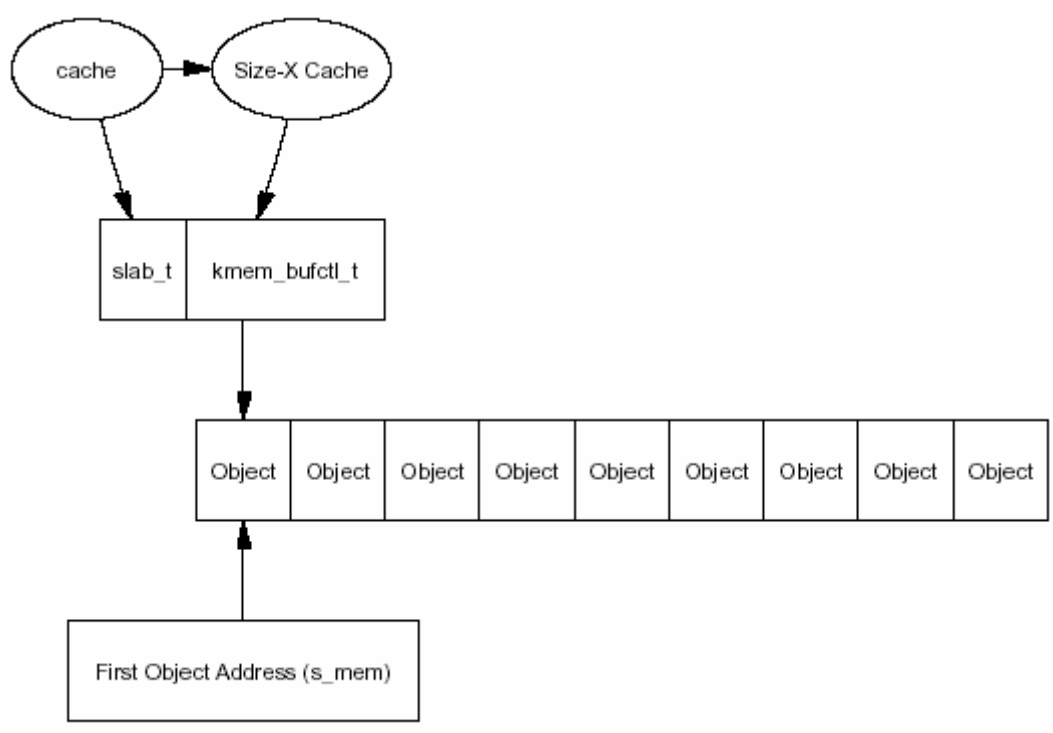
캐쉬들은 next 필드로 링크드 리스트를 구성한다. 각각의 캐쉬는 하나 이상의 슬랩으로 이뤄지고 각각의 슬랩은 하나 이상의 페이지로 이뤄진다. 각각의 슬랩에는 여러 개의 객체들이 저장되는데 하드웨어 캐쉬에 맞추기위해 각 객체들 사이에 빈 공간을 둘 수도 있다. 예를 들어 객체의 크기가 30바이트인데 32바이트 정렬을 위해서라면 객체 뒤에 2바이트씩 틈을 만드는 것이다.

다음 그림은 내부 슬랩을 나타낸 것이다. 내부 슬랩에서는 슬랩의 시작 부분에 슬랩의 디스크립터가 존재한다.




[ 그림 3-7 ]

다음 그림은 외부 슬랩을 나타낸 것이다. 외부 슬랩에서는 cache_sizes에서 슬랩의 디스크립터를 저장할 메모리 공간을 얻어온다.



[ 그림 3-8 ]

### 3.2.1 kmem_cache_slabmgmt()

이 함수는 slab_t를 저장할 메모리를 할당받아서 캐쉬에 추가한다. kmem_cache_grow()에서 호출된다.

```
/* Get the memory for a slab management obj. */
static inline slab_t * kmem_cache_slabmgmt (kmem_cache_t *cachep,
                        void *objp, int colour_off, int local_flags)
{
```

- cachep 슬랩을 추가할 캐쉬
- objp 슬랩을 위해 할당된 페이지의 시작 수조
- colour_off 슬랩에 적용될 colour
- local_flags 슬랩에 적용된 속성

```
        slab_t *slabp;

        if (OFF_SLAB(cachep)) {
                /* Slab management obj is off-slab. */
                slabp = kmem_cache_alloc(cachep->slabp_cache, local_flags);
                if (!slabp)
                        return NULL;
```

- 만약 외부 슬랩을 사용한다면 캐쉬를 생성하는 kmem_cache_create() 함수에서 cachep->slabp_cache 항목을 다음과 같이 초기화한다.

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

- kmem_find_general_cachep() 함수는 cache_sizes 에서 슬랩을 저장할 메모리를 얻어오는 함수이다.
- 결국 외부 슬랩을 저장할 메모리를 할당해서 cachep->slabp_cache 변수에 저장하는 것이다.
- kmem_cache_alloc() 함수와 kmem_find_general_cachep() 함수는 다음 장에서 설명한다.

```
        } else {
                /* FIXME: change to
                        slabp = objp
                 * if you enable OPTIMIZE
                 */
                slabp = objp+colour_off;
                colour_off += L1_CACHE_ALIGN(cachep->num *
                                sizeof(kmem_bufctl_t) + sizeof(slab_t));
        }
```

- 내부 슬랩을 사용한다면 슬랩 디스크립터의 시작 주소는 페이지의 시작 부분에서 colour_off만큼 떨어진 곳이 된다. colour_off 값은 kmem_cache_grow()에서 계산된 값으로 현재 슬랩이 사용할 colour 값이 된다.
- 이제 colour_off 변수를 객체들이 시작될 주소로 바꾼다. 객체들은 슬랩과 객체들을 관리한 kmem_bufctl_t 데이터들 뒤에 위치하게되므로 데이터들의 크기와 colour_off 값을 더하면 된다.

```
        slabp->inuse = 0;
```

- 사용중인 객체는 없다

```
        slabp->colouroff = colour_off;
```

- 슬랩의 colour offset 값을 저장한다.

```
        slabp->s_mem = objp+colour_off;
```

- s_mem은 첫번째 객체의 시작 주소를 말한다. 즉 페이지의 시작 주소 objp에 슬랩의 colour offset 값을 더하면 된다.

```
        return slabp;
}
```

- 슬랩의 포인터를 반환한다.

### 3.2.2 kmem_find_general_cachep()

cache_sizes에서 원하는 크기 만큼의 메모리를 찾아서 반환한다.

```
kmem_cache_t * kmem_find_general_cachep (size_t size, int gfpflags)
```

```
{
        cache_sizes_t *csizep = cache_sizes;

        /* This function could be moved to the header file, and
         * made inline so consumers can quickly determine what
         * cache pointer they require.
         */
        for ( ; csizep->cs_size; csizep++) {
                if (size > csizep->cs_size)
                        continue;
                break;
        }
        return (gfpflags & GFP_DMA) ? csizep->cs_dmacachep : csizep->cs_cachep;
}
```

### 3.3 objects관련 함수

사실상 슬랩 할당자에서 중요하게 사용되는 함수들은 캐쉬와 슬랩을 다루는 함수이며 객체를 다루는 함수는 매우 단순하다. 따라서 핵심되는 함수 몇 가지만을 분석한다.

#### 3.3.1 kmem_cache_init_objs()

객체의 생성자를 호출해서 모든 객체들을 초기화하는 함수이다. kmem_cache_grow()함수에서 슬랩이 생성될 때 호출되며 리눅스에서는 생성자를 사용하지 않으므로 디버깅 코드를 제외하면 거의 하는 일이 없다.

```
static inline void kmem_cache_init_objs (kmem_cache_t * cachep,
                        slab_t * slabp, unsigned long ctor_flags)
{
        int i;

        for (i = 0; i < cachep->num; i++) {
                void* objp = slabp->s_mem+cachep->objsize*i;
```

```
                /*
                 * Constructors are not allowed to allocate memory from
                 * the same cache which they are a constructor for.
                 * Otherwise, deadlock. They must also be threaded.
                 */
                if (cachep->ctor)
                        cachep->ctor(objp, cachep, ctor_flags);
```

- 만약 생성자가 있다면 생성자를 호출해서 객체를 초기화한다.

```
                slab_bufctl(slabp)[i] = i+1;
```

- kmem_bufctl_t는 다음과 같이 정의되어있다.

```
typedef unsigned int kmem_bufctl_t;
```

- 결국 여러 개의 객체를 관리하기 위한 이 데이터는 객체들의 인덱스로 사용되는 정수 값이다.
- slab_bufctl 매크로는 slab.c 파일에 다음과 같이 정의되어있다.

```
#define slab_bufctl(slabp) ₩
    ((kmem_bufctl_t *)(((slab_t*)slabp)+1))
```

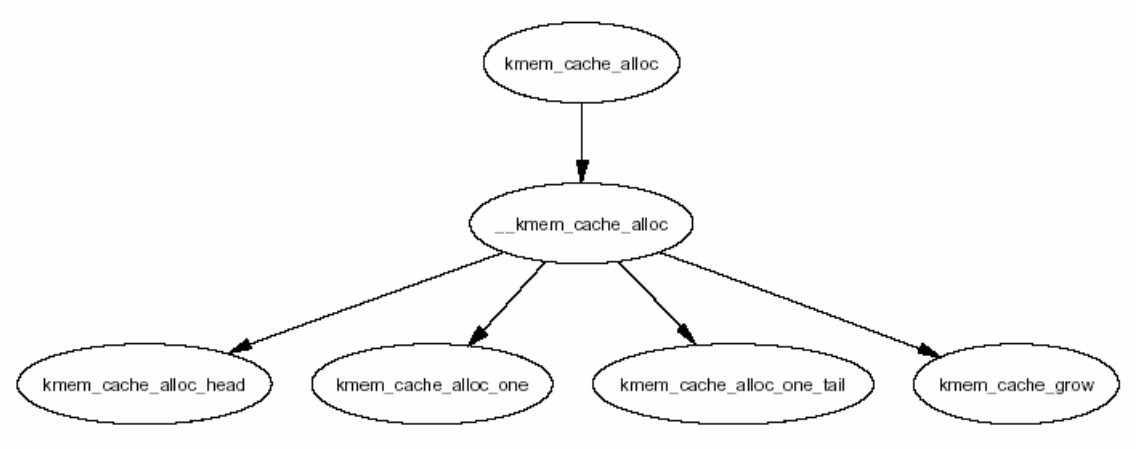
- slabp는 슬랩 디스크립터의 주소이다. (slab_t *)slabp+1은 slabp의 바로 다음 주소를 가리키게 된다. 이것을 (kmem_bufctl_t *)로 캐스팅하면 결국 객체를 관리하기 위한 kmem_bufctl_t 배열의 시작 주소가 되는 것이다.
- 슬랩 디스크립터의 바로 다음에는 kmem_bufctl_t의 배열이 저장된다. 이 데이터는 다음 빈 객체를 가리키는데 사용되는데 현재 객체들을 초기화할 때는 모두 빈 객체들이므로 다음 객체의 인덱스들을 저장하게된다.

```
        }
        slab_bufctl(slabp)[i-1] = BUFCTL_END;
        slabp->free = 0;
}
```

- 마지막 배열에는 -1 값을 저장해서 마지막이라는 것을 나타내주고 free를 0으로 저장해서 객체를 할당해줄 때 0번째 객체를 반환해주도록 한다.

### 3.3.2 kmem_cache_alloc()

객체를 할당하는 것은 유니 프로세서 환경과 멀티 프로세서 환경이 서로 다르다. 여기서는 유니 프로세서 환경만 다루도록 한다. 다음 그림은 유니 프로세서 환경에서 객체를 할당하기 위한 함수들의 연관 관계를 보여준다.




[ 그림 3-9 ]

객체를 할당하는 데는 크게 네가지 단계가 있다.

① 할당이 허용되는지 기본적인 검사를 한다.

② 어떤 슬랩에서 할당을 해야하는지 선택한다. slabs_partial 와 slabs_free가 모두 해당될 수 있다.

③ 만약 slabs_free에 아무 슬랩도 있지 않다면 캐쉬에 새로운 슬랩을 추가해 줘야 한다(kmem_cache_grow()함수 참조).

④ 골라진 슬랩에서 객체를 반환한다.

함수는 단순히 __kmem_cache_alloc() 함수만을 호출한다.

```
/**
 * 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);
}
```

### 3.3.3 __kmem_cache_alloc()

이 함수는 두 개의 인자를 가진다.

kmem_cache_t *cachep 할당이 일어날 캐쉬

int flags 할당을 위한 플래그들

플래그들은 include/linux/slab.h에서 정의되어있고 주로 할당을 나타내는데 사용된다. 단일 프로세스 환경에서의 코드는 다음과 같다.

```
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 하게 된다.

```
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;
}
```

### 3.3.4 MACRO kmem_cache_alloc_one()

함수가 아니고 매크로로 만들어졌다. __kmem_cache_alloc() 함수에는 alloc_new_slab: 으로 점프하는 코드가 숨어있는데 이 매크로안에 있기 때문이다.
빈 객체를 가진 슬랩을 찾아서 kmem_cache_alloc_one_tail() 함수를 호출한다.

```
/*
 * 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 것이

다. 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.

})

### 3.3.5 kmem_cache_alloc_one_tail()

빈 객체를 가진 슬랩이 찾아지면 호출되는 함수이다. 디버깅을 위한 코드를 제외하면 간단한 함수이다.

```
static inline void * kmem_cache_alloc_one_tail (kmem_cache_t *cachep, slab_t *slabp)
{
    void *objp;

    STATS_INC_ALLOCED(cachep);
    STATS_INC_ACTIVE(cachep);
    STATS_SET_HIGH(cachep);
```

- 캐쉬 디스크립터에 객체의 사용에 대한 정보를 갱신해준다.

```
    /* get obj pointer */
    slabp->inuse++;
    objp = slabp->s_mem + slabp->free*cachep->objsize;
    slabp->free=slab_bufctl(slabp)[slabp->free];
```

- s_mem은 객체가 저장되기 시작되는 메모리의 주소이다. free는 슬랩안에

있는 첫번째 빈 객체의 인덱스이다. 따라서 free와 객체의 크기를 곱하면 빈 객체의 주소를 얻을 수 있다.

- 현재 할당된 객체를 가르키던 kmem_bufctl_t 변수에는 이 객체 다음에 위치한 빈 객체의 인덱스가 저장되어있다. 따라서 이 값으로 슬랩 디스크립터의 free 변수를갱신한다.

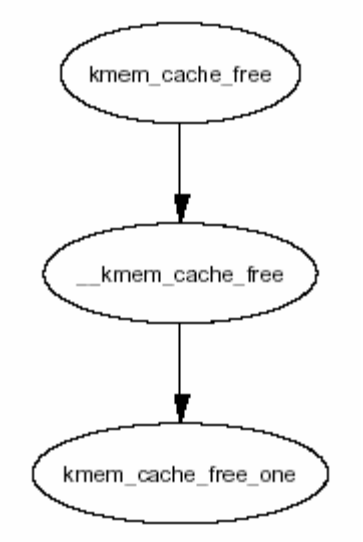
```
        if (unlikely(slabp->free == BUFCTL_END)) {
                list_del(&slabp->list);
                list_add(&slabp->list, &cachep->slabs_full);
        }
```

- 만약 할당된 객체가 마지막 빈 객체였다면 슬랩을 slabs_full 리스트로 옮긴다.

```
        return objp;
}
```

### 3.3.6 kmem_cache_free()

객체를 해제하는 것은 할당과 매우 비슷하다. 다음은 객체를 해제하는데 사용되는 함수들의 순서이다.




[ 그림 3-10 ]

kmem_cache_free()함수는 디버깅 코드를 제외하면 단순히 __kmem_cache_free() 함수를 호출하는 일을 한다.

```
/**
 * kmem_cache_free - Deallocate an object
 * @cachep: The cache the allocation was from.
 * @objp: The previously allocated object.
 *
 * Free an object which was previously allocated from this
 * cache.
 */
void kmem_cache_free (kmem_cache_t *cachep, void *objp)
{
        unsigned long flags;
        local_irq_save(flags);
        __kmem_cache_free(cachep, objp);
        local_irq_restore(flags);
}
```

### 3.3.7 __kmem_cache_free()

유니 프로세서 환경에서는 곧바로 kmem_cache_free_one() 함수를 호출하고 종료된다.

```
/*
 * __kmem_cache_free
 * called with disabled ints
 */
static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp)
{
        kmem_cache_free_one(cachep, objp);
}
```

### 3.3.8 kmem_cache_free_one()

```
static inline void kmem_cache_free_one(kmem_cache_t *cachep, void *objp)
{
```

```
slab_t* slabp;

slabp = GET_PAGE_SLAB(virt_to_page(objp));
```

- virt_to_page() 함수는 인자로 넘긴 주소를 포함하는 페이지의 디스크립터를 반환한다. 페이지 디스크립터의 list.prev 항목이 슬랩의 포인터를 가지고 있으므로 슬랩의 주소를 알아낼 수 있다.

```
{
        unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
```

- 해제된 객체의 인덱스를 계산한다.

```
        slab_bufctl(slabp)[objnr] = slabp->free;
        slabp->free = objnr;
}
```

- 해제된 객체에 대응되는 kmem_bufctl_t 항목의 값을 현재 객체를 해제하기 전에 첫번째 빈 객체였던 객체의 인덱스로 갱신한다. 그리고 이제 첫번째 빈 객체의 인덱스는 현재 해제되고 있는 객체의 인덱스가 된다.

```
STATS_DEC_ACTIVE(cachep);

/* fixup slab chains */
{
        int inuse = slabp->inuse;
        if (unlikely(!--slabp->inuse)) {
                /* Was partial or full, now empty. */
                list_del(&slabp->list);
                list_add(&slabp->list, &cachep->slabs_free);
```

- 사용중인 객체의 수가 0이 됐다면 slabs_free 리스트로 슬랩을 옮긴다.

```
        } else if (unlikely(inuse == cachep->num)) {
                /* Was full. */
                list_del(&slabp->list);
                list_add(&slabp->list, &cachep->slabs_partial);
        }
```

- 감소하기 이전의 사용중인 객체의 수가 캐쉬가 가질 수 있는 최대 객체의 수였다면 slabs_partial 리스트로 옮긴다.

- 두 상황이 모두 아니라면 slabs_partial에 연결되어있고 아직 사용중인 객체가 몇 개 있다는 뜻이므로 리스트에 그대로 놔둔다.
}
}

## 3.4 자유 객체의 관리

슬랩 할당자는 슬랩에 속해있는 객체들을 관리하기 위한 좋은 방법을 가지고 있다. 자유 객체가 어디 있는지 매우 간편하게 알아낼 수 있는데 kmem_bufctl_t 데이터를 이용하는 방법이 그것이다. 처음에 슬랩 할당자를 개발했던 논문(Jeff Bonwick. The Slab Allocator: An Object Caching Kernel Memory Allocator) 에서 kmem_bufctl_t는 객체들의 리스트였다. 하지만 2.4.X 버전의 리눅스 커널에서는 unsigned int 형의 데이터이며 객체들의 인덱스를 저장하게된다. 인덱스를 가지고도 충분히 객체들을 관리할 수 있기 때문이다.

### 3.4.1 kmem_bufctl_t

kmem_bufctl_t 데이터 형은 단순히 unsigned int 형이며 슬랩 디스크립터 바로 뒤에 연속된 배열로 저장된다. 슬랩에 들어있는 객체의 수만큼 배열을 이루게 된다.

```
typedef unsigned int kmem_bufctl_t;
```

이 배열의 시작 주소는 따로 저장되거나 기억되지 않는다. 대신 slab_bufctl 매크로로 이 배열의 시작 주소를 계산한다.

```
#define slab_bufctl(slabp) ₩
        ((kmem_bufctl_t *)(((slab_t*)slabp)+1))
```

((slab_t *)slabp+1) 는 slabp의 주소를 slab_t 데이터 형으로 캐스트하고 1을 더해서 slabp의 주소에 slab_t 데이터형의 크기만큼 더한 값을 나타내게 해준다. 이 주소값을 (kmem_bufctl_t *) 형으로 캐스트하면 결국 슬랩 디스크립터의 바로 뒤에 저장되는 kmem_bufctl_t 데이터 배열의 시작 주소를 나타내게 되는 것이다. 따라서 slab_bufctl(slabp)[i] 와 같은 코드는 이 kmem_bufctl_t 배열에서 i번째 항목을 가르키게 되는 것이다.

kmem_bufctl_t 데이터는 마치 리스트에서 다음 항목의 포인터를 저장하는 것처럼 다음 빈 객체의 인덱스를 저장하고있다. 첫번째 빈 객체의 인덱스는 슬랩 디스크립

터의 free 항목이 저장하고 있으므로 slab_bufctl(slabp)[free] 와 같은 코드는 kmem_bufctl_t 배열에서 free번째 항목의 값이므로 free 다음의 빈 객체에 대한 인덱스를 가지고 있는 것이다. 결국 이런 식으로 인덱스의 값을 가지고 리스트처럼 빈 객체들을 할당하는데 사용한다.

### 3.4.2 kmem_bufctl_t 배열의 초기화

캐쉬에 새로운 슬랩을 추가할 때마다 슬랩의 모든 객체와 kmem_bufctl_t 배열은 초기화된다. 이 배열의 각 항목에는 1부터 BUFCTL_END의 값이 저장된다. 또 slab_t->free에는 0이 저장된다. 이것은 첫번째 빈 객체의 인덱스는 0이고 0 다음에는 1, 다음에는 2…가 된다는 것이고 마지막 항목에는 더 이상의 객체가 없다는 것을 알려주는 것이다. 즉 n번째 객체의 다음 객체는 kmem_bufctl_t[n] 이라는 의미가 된다.

### 3.4.3 빈 객체 찾기

kmem_cache_alloc() 함수는 새 객체를 할당해준다. 이 함수는 kmem_cache_alloc_one_tail() 함수를 호출하고 여기서 객체의 할당과 kmem_bufctl_t 배열의 처리가 이뤄진다.
슬랩 디스크립터의 free 항목이 첫번째 빈 객체의 인덱스를 가지고있다. 다음으로 빈 객체의 인덱스는 kmem_bufctl_t[free]가 된다. 코드로 나타내면 다음과 같다.

```
objp = slabp->s_mem + slabp->free*cachep->objsize;
slabp->free=slab_bufctl(slabp)[slabp->free];
```

s_mem은 객체가 저장되기 시작되는 메모리의 주소이다. free는 슬랩안에 있는 첫번째 빈 객체의 인덱스이다. 따라서 free와 객체의 크기를 곱하면 빈 객체의 주소를 얻을 수 있다. 현재 할당된 객체를 가르키던 kmem_bufctl_t 변수에는 이 객체 다음에 위치한 빈 객체의 인덱스가 저장되어있다. 따라서 이 값으로 슬랩 디스크립터의 free 변수를갱신한다.
kmem_bufctl_t[free]에서 kmem_bufctl_t의 시작 주소를 따로 저장하지 않았으므로 slab_bufctl(slabp) 매크로를 이용해서 알아내는 것이다.

### 3.4.4 kmem_bufctl_t 의 갱신

객체를 해제하였을 때 빈 객체가 새로 들어왔으므로 kmem_bufctl_t 배열을 갱신

해야한다. kmem_cache_free_one() 함수에서 다음의 코드가 이 일을 한다.

```
unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
slab_bufctl(slabp)[objnr] = slabp->free;
slabp->free = objnr;
```

objp는 이제 해제될 객체의 주소이고 objnr은 그 객체의 인덱스가 된다. 이전에 저장되어 있던 첫번째 빈 객체의 주소는 이제 kmem_bufctl_t[objnr]에 저장되고 free에는 objnr이 저장되면 갱신이 완료된다.

## 3.5 Buddy Allocator 시스템과의 연동

슬랩 할당자에는 직접 페이지를 다루는 부분이 없다. 이 부분은 버디 할당자와 같은 물리적인 페이지의 할당을 다루는 다른 서브 시스템이 맞게된다. 이 두 시스템 사이에는 두가지 인터페이스가 있다.

### 3.5.1 kmem_getpages()

캐쉬가 필요로 하는 페이지 수를 가지고 __get_free_pages 함수를 호출한다.

```
/* Interface to system's page allocator. No need to hold the cache-lock.
 */
static inline void * kmem_getpages (kmem_cache_t *cachep, unsigned long flags)
{
        void    *addr;

        /*
         * If we requested dmaable memory, we will get it. Even if we
         * did not request dmaable memory, we might get it, but that
         * would be relatively rare and ignorable.
         */
        flags |= cachep->gfpflags;
        addr = (void*) __get_free_pages(flags, cachep->gfporder);
        /* Assume that now we have the pages no one else can legally
         * messes with the 'struct page's.
         * However vm_scan() might try to test the structure to see if
```

```
         * it is a named-page or buffer-page.  The members it tests are
         * of no interest here.....
         */
        return addr;
}
```

### 3.5.2 kmem_freepages()

```
 /* Interface to system's page release. */
static inline void kmem_freepages (kmem_cache_t *cachep, void *addr)
{
        unsigned long i = (1<<cachep->gfporder);
        struct page *page = virt_to_page(addr);

        /* free_pages() does not clear the type bit - we do that.
         * The pages have been unlinked from their cache-slab,
         * but their 'struct page's might be accessed in
         * vm_scan(). Shouldn't be a worry.
         */
        while (i--) {
                PageClearSlab(page);
                page++;
        }
```

- 각 페이지의 디스크립터에 저장된 슬랩에 관한 정보들을 제거한다.

```
        free_pages((unsigned long)addr, cachep->gfporder);
```

- 버디 알고리즘에게 페이지들을 반환한다.

```
}
```

## 3.6 cache_sizes

리눅스에는 크기가 작은 메모리의 할당을 위해 미리 정해진 크기들의 객체들을 가지고있는 캐쉬를 관리한다. 이 캐쉬는 같은 크기의 DMA를 위한 메모리와 아닌 메모리를 모두 관리한다. cache_sizes_t 구조체가 이 캐쉬를 나타낸다.

```
/* Size description struct for general caches. */
typedef struct cache_sizes {
        size_t             cs_size;
        kmem_cache_t *cs_cachep;
        kmem_cache_t *cs_dmacachep;
} cache_sizes_t;

static cache_sizes_t cache_sizes[] = {
#if PAGE_SIZE == 4096
        {      32,         NULL, NULL},
#endif
        {      64,         NULL, NULL},
        {     128,         NULL, NULL},
        {     256,         NULL, NULL},
        {     512,         NULL, NULL},
        {    1024,         NULL, NULL},
        {    2048,         NULL, NULL},
        {    4096,         NULL, NULL},
        {    8192,         NULL, NULL},
        {   16384,         NULL, NULL},
        {   32768,         NULL, NULL},
        {   65536,         NULL, NULL},
        {  131072,         NULL, NULL},
        {       0,         NULL, NULL}
};
```

- cs_size 메모리 블록의 크기
- cs_cachep 일반 메모리 블록을 위한 캐쉬
- ca_dmacachep DMA용 메모리 블록을 위한 캐쉬

kmem_cache_sizes_init() 함수가 이 캐쉬의 초기화를 담당한다. 32바이트부터 128K 까지의 크기로 캐쉬들을 생성한다.

### 3.6.1 kmalloc()

이 함수는 커널이 작은 메모리 버퍼들을 요구할 때 사용한다. 특정한 크기의 메모

리를 요구하면 이에 맞는 크기의 객체를 가진 캐쉬에서 객체를 할당해준다.

```
/**
 * kmalloc - allocate memory
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate.
 *
 * kmalloc is the normal method of allocating memory
 * in the kernel.
 *
 * The @flags argument may be one of:
 *
 * %GFP_USER - Allocate memory on behalf of user.  May sleep.
 *
 * %GFP_KERNEL - Allocate normal kernel ram.  May sleep.
 *
 * %GFP_ATOMIC - Allocation will not sleep.  Use inside interrupt handlers.
 *
 * Additionally, the %GFP_DMA flag may be set to indicate the memory
 * must be suitable for DMA.  This can mean different things on different
 * platforms.  For example, on i386, it means that the memory must come
 * from the first 16MB.
 */
void * kmalloc (size_t size, int flags)
{
        cache_sizes_t *csizep = cache_sizes;

        for (; csizep->cs_size; csizep++) {
                if (size > csizep->cs_size)
                        continue;
```

- 요청된 크기와 비슷하면서 큰 객체를 가진 캐쉬를 찾는다.

```
                return __kmem_cache_alloc(flags & GFP_DMA ?
                         csizep->cs_dmacachep : csizep->cs_cachep, flags);
```

- DMA를 위한 메모리인지 아닌지를 판단해서 적당한 메모리를 반환해준다.

```
        }
        return NULL;
}
```

### 3.6.2 kfree()

kmalloc()에서 얻는 작은 메모리 블록을 해제하는데 사용된다. 해제될 메모리가 속한 캐쉬를 알아내서 __kmem_cache_free() 함수를 호출하는 일을 한다.

```
/**
 * kfree - free previously allocated memory
 * @objp: pointer returned by kmalloc.
 *
 * Don't free memory not originally allocated by kmalloc()
 * or you will run into trouble.
 */
void kfree (const void *objp)
{
        kmem_cache_t *c;
        unsigned long flags;

        if (!objp)
                return;
        local_irq_save(flags);
        CHECK_PAGE(virt_to_page(objp));
        c = GET_PAGE_CACHE(virt_to_page(objp));
        __kmem_cache_free(c, (void*)objp);
        local_irq_restore(flags);
}
```

# PART (II) BUDDY SYSTEM

## 1 버디 시스템 개론

버디 시스템은 외부 단편화를 최대한 줄이고 페이지의 할당과 해제의 속도를 높이기 위해 개발된 메모리 할당 알고리즘이다. 외부 단편화를 줄이기 위해 연속된 자유 페이지들은 크기에 따라 리스트로 묶여서 관리된다. 연속된 2개의 페이지들의 리스트, 연속된 4개의 페이지들의 리스트 등으로 2의 배수로 연속된 페이지들의 리스트를 관리하면서 필요한 양의 페이지의 수에 맞는 리스트에서 페이지를 할당해 주어서 외부 단편화를 줄일 수가 있다. 만약 요청과 동일한 크기의 연속된 페이지가 없다면 그 다음으로 큰 페이지들을 반으로 나눠서 반은 할당하고 나머지 반은 아래 리스트에 연결하는 방식으로 동작한다. 할당되는 페이지들의 시작 주소는 항상 페이지 블록 크기의 배수가 된다. 예를 들어 8개의 페이지를 할당한다면 첫번째 페이지의 인덱스는 8의 배수가 된다는 것이다.

반대로 어떤 갯수의 페이지들을 해제할 때면 같은 갯수이고 서로 연결되는 자유 페이지들이 있는지 확인해서 두 쌍을 합친다. 이렇게 주소가 연결되며 같은 크기의 페이지 블록들을 버디(buddy)라고 부르기 때문에 버디 시스템이라고 부른다. 이런 병합 작용은 계속 더 큰 크기의 블록으로 반복되서 실행되서 최대 크기의 블록으로 유지된다.

## 2 버디 시스템에서 사용하는 데이터

버디 시스템에서는 물리 페이지를 관리하는 페이지 디스크립터와 페이지들의 리스트를 관리하는 리스트와 비트맵등의 데이터를 사용한다.

### 2.1 페이지 디스크립터

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page.
 *
```

```
 * Try to keep the most commonly accessed fields in single cache lines
 * here (16 bytes or greater).  This ordering should be particularly
 * beneficial on 32-bit processors.
 *
 * The first line is data used in page cache lookup, the second line
 * is used for linear searches (eg. clock algorithm scans).
 *
 * TODO: make this structure smaller, it could be as small as 32 bytes.
 */
typedef struct page {
        struct list_head list;          /* ->mapping has some page lists. */
        struct address_space *mapping;      /* The inode (or ...) we belong to.
*/
        unsigned long index;            /* Our offset within mapping. */
        struct page *next_hash;             /* Next page sharing our hash
bucket in
                                            the pagecache hash table. */
        atomic_t count;                     /* Usage count, see below. */
        unsigned long flags;            /* atomic flags, some possibly
                                            updated asynchronously */
        struct list_head lru;           /* Pageout list, eg. active_list;
                                            protected by pagemap_lru_lock !! */
        unsigned long age;              /* Page aging counter. */
        wait_queue_head_t wait;             /* Page locked?  Stand in line... */
        struct page **pprev_hash;       /* Complement to *next_hash. */
        struct buffer_head * buffers;   /* Buffer maps us to a disk block. */
        void *virtual;                  /* Kernel virtual address (NULL if
                                            not kmapped, ie. highmem) */
        struct zone_struct *zone;       /* Memory zone we are in. */
} mem_map_t;
```

몇가지 중요한 항목들만 분석한다.

- count 페이지를 참조하고있는 프로세스의 수. 빈 페이지라면 0이 된다.
- list 디스크립터의 리스트에서 사용
- flags 페이지 프레임의 상태를 알려주는 플래그. PG_xyz라는 이름의 플래그들이 있으며 PageXyz 매크로로 플래그의 상태를 읽거나 쓸 수 있다.

모든 페이지 프레임의 디스크립터가 저장된 배열이 mem_map 이다. 디스크립터의 크기가 64바이트 이하이므로 1M의 메모리를 관리하는 디스크립터를 위해 4페이지의 메모리가 필요하다.

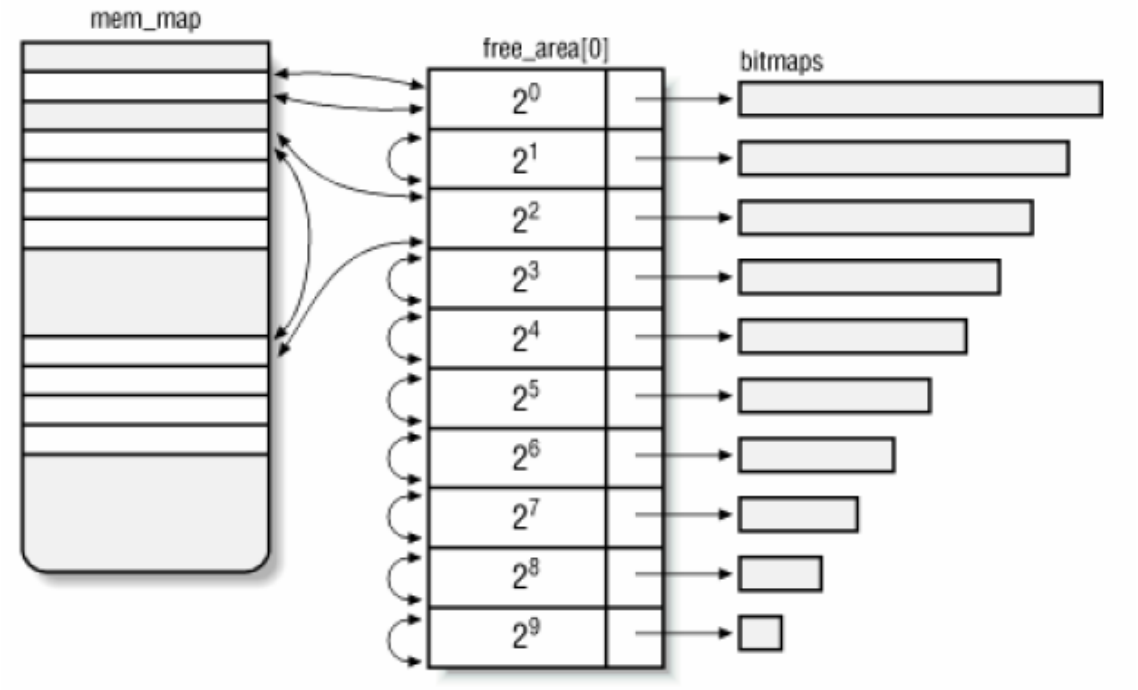
## 2.2 free_area_struct

리눅스는 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 페이지 블록의 리스트를 사용한다. 이 리스트를 관리하고 버디 시스템을 구현하기 위해서 free_area_struct 구조체를 사용한다.

```
typedef struct free_area_struct {
        struct list_head free_list;
        unsigned long           *map;
} free_area_t;
```

각 항목에 대한 설명이다.

- free_list 특정 크기의 자유 페이지 블록들의 이중 연결 리스트이다. 페이지 블록에서 첫번째의 페이지에 대한 디스크립터를 가르키고 이 디스크립터는 다음 페이지 블록을 가르키는 형식으로 연결 리스트가 형성된다. 즉 다음 그림과 같다.



[ 그림 2-1 ]

4개의 페이지 블록의 리스트를 보면 free_area_t에서 첫번째 페이지를 가르키고있고 그 페이지는 또 다음 페이지 블록의 첫 페이지를 가르키고 또 그 페이지는 다시 free_area_t를 가르키게되서 이중 원형 리스트가 형성된다.

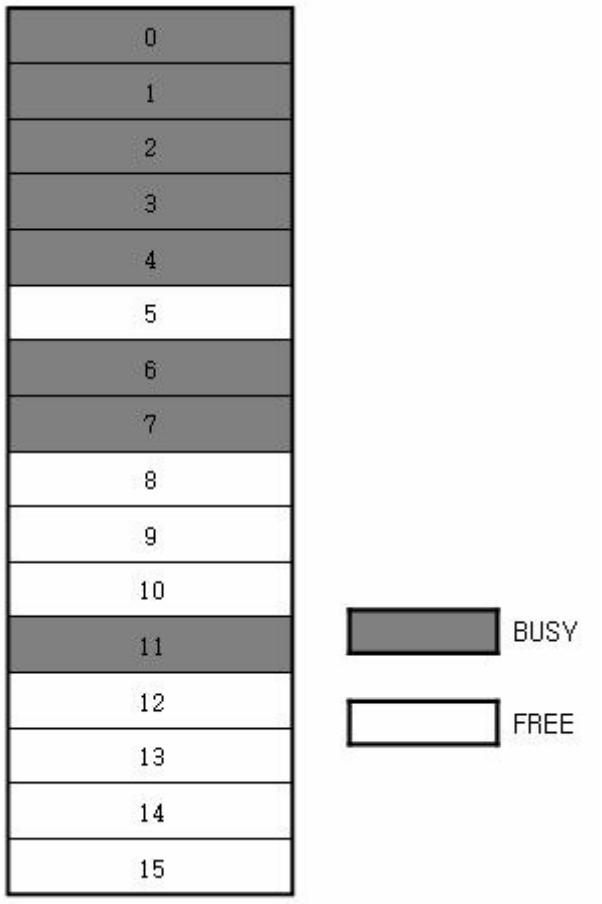
- map 버디 쌍의 상태가 어떤지를 나타내는 비트맵을 가르킨다. 이 비트맵의 크기는 다음 공식으로 계산할 수 있다.

((number of pages) - 1) >> (order + 4)) + 1 bytes

한 비트가 하나의 버디의 상태를 나타낸다. 2의 2제곱의 크기를 가진 버디들의 비트맵에서 16번 페이지로 시작하는 페이지 블록이 몇 번째 비트로 나타나는지 계산해보자. 우선 한 비트는 버디 쌍을 나타내고 한 버디는 두 개의 페이지 블록이므로 결국 한 비트는 8개의 페이지를 나타낸다. 따라서 0번 비트는 0~7번 페이지를 나타내고 1번 비트는 8~15번 페이지를 나타낸다. 결국 16번 페이지로 시작하는 페이지 블록은 2번 비트를 읽으면 상태를 알 수 있다. 식으로 나타내면 다음과 같다.

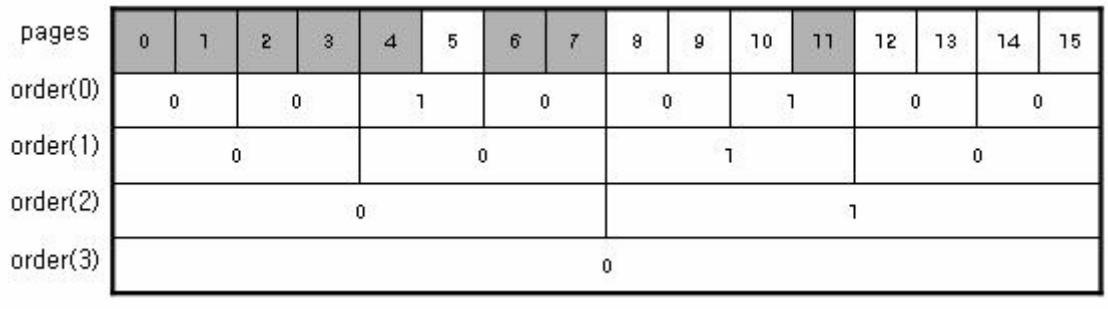
(index of first page) >> (size of order + 1)

# 3 버디 시스템의 예제




[ 그림 3-1 ]

16개의 페이지가 있고 0,1,2,3,4,6,7,11 페이지는 사용중이고 나머지는 비어있다고 하면 비트맵은 다음과 같이 된다.



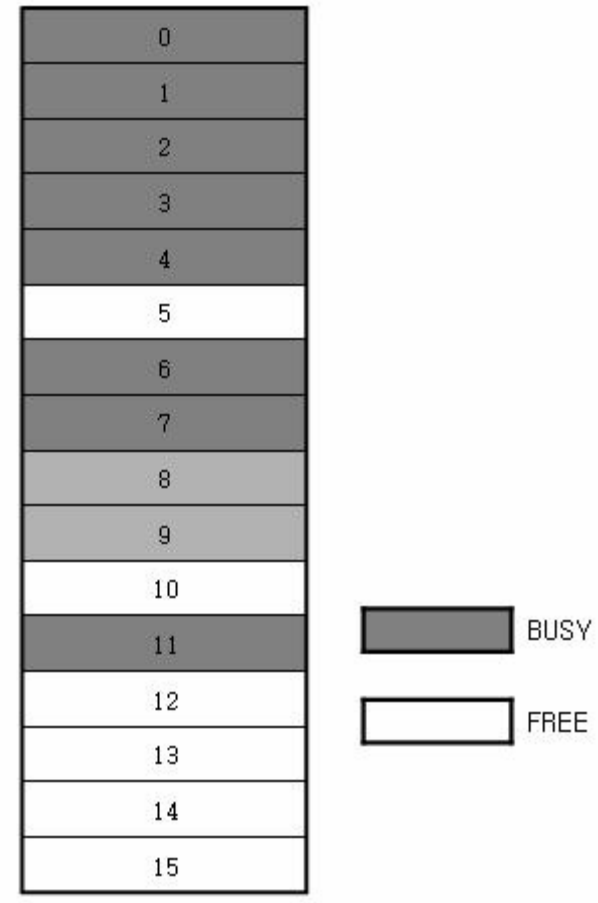

[ 그림 3-2 ]

3.1 할당

1. order(1) 즉 2페이지를 할당하는 과정을 생각해본다.
2. free list는 다음과 같다.

order(0) : 5, 10
order(1) : 8 [8,9]
order(2) : 12 [12, 13, 14, 15]
order(3) :

3. 마침 order(1)에 비어있는 블록이 있으므로 8번 페이지의 포인터를 반환하고 free list에서 삭제한다.




[ 그림 3-3 ]

4. 또다시 order(1)의 블록이 필요하면 우선 order(1)의 리스트를 검색해본다.
5. 비어있는 블록이 없으므로 좀더 큰 블록을 검색한다. order(2)를 검색한다.
6. 12번 페이지부터 비어있는 블록이 있다. 이 블록은 이제 [12,13]과

[14,15] 두 블록으로 나눠진다.

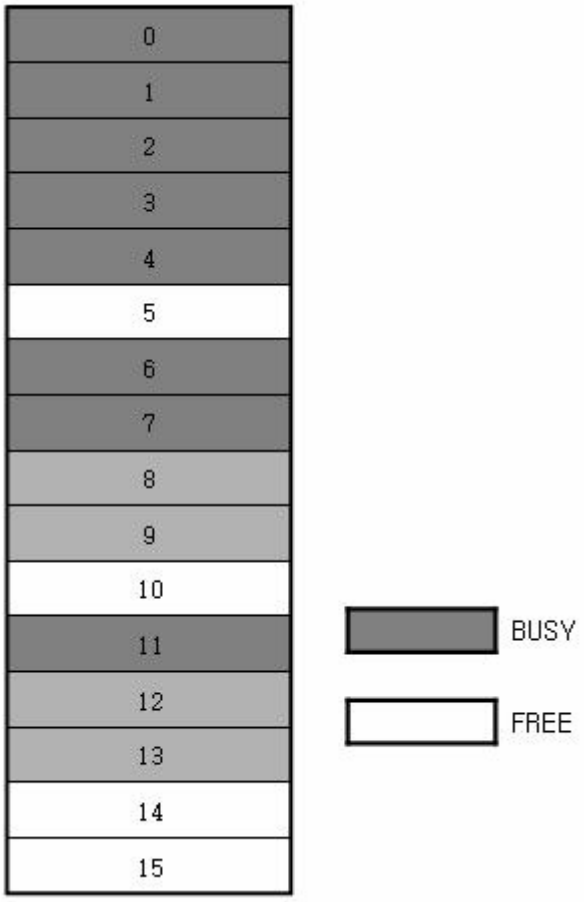
7. [14,15] 블록은 order(2)에서 order(1)로 내려가서 free list로 연결된다.
8. [12,13] 블록의 포인터는 반환된다.
9. 최종적으로 free list는 다음과 같이 된다.

order(0) : 5, 10

order(1) : 14 [14,15]

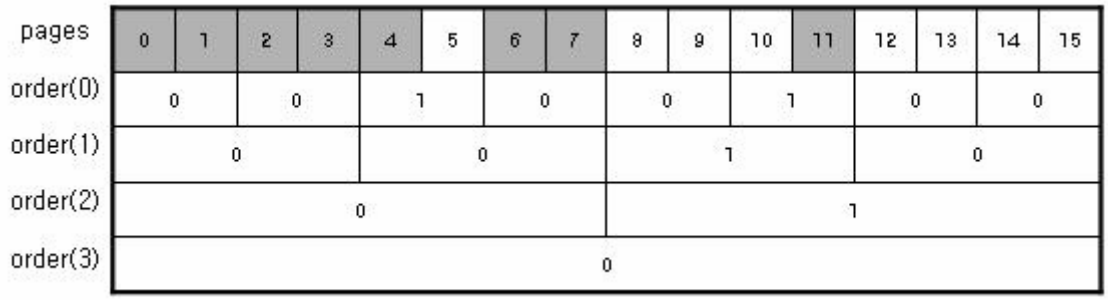
order(2) :

order(3) :




[ 그림 3-4 ]

## 3.2 해제




[ 그림 3-5 ]

1. 다시 같은 예제를 가지고 시작한다. 자유 리스트는 다음과 같다.

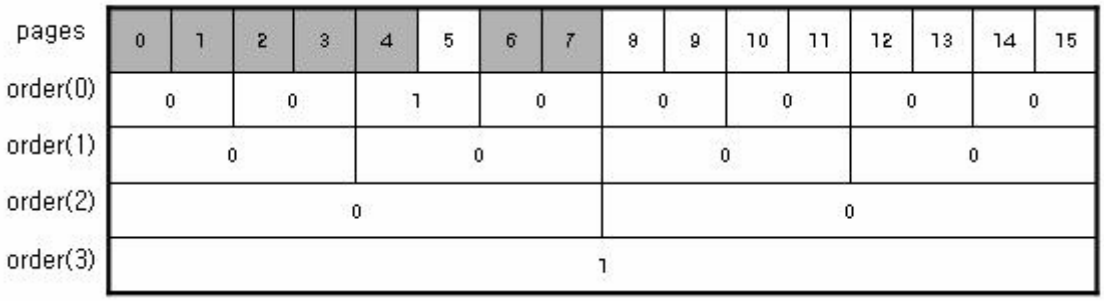
   order(0) : 5, 10
   order(1) : 8 [8,9]
   order(2) : 12 [12, 13, 14, 15]
   order(3) :

2. 11번 페이지를 해제한다면 11번 페이지에 해당하는 비트를 찾아야 한다.
3. 11번 페이지 하나를 해제하므로 order(0) 리스트에서 찾아야한다. 다음과 같은 식으로 비트를 찾을 수 있다.

   index = page_idx >> (order + 1)
   = 11 >> (0 + 1)
   = 5

4. 하나의 비트가 2개의 페이지 블록을 가르키므로 하나의 비트가 2^(order+1)개의 페이지에 매핑된다. 따라서 원하는 페이지가 몇번째 비트에 매핑되는지 알아내기 위해서는 page_idx / 2^(order+1)이 되고 따라서 page_idx>>(order+1)이 된다.
5. 즉 5번 비트 (6번째 비트)가 11번 페이지를 가르킨다.
6. 비트맵에서 비트가 1이면 그 페이지 블록의 버디는 비어있는 페이지 블록이라는 뜻이다. 5번 비트가 1이므로 10번 페이지는 비어있다는 뜻이 된다.
7. 따라서 두 버디가 모두 비어있게되므로 5번 비트는 0이된다.
8. 이제 [10,11] 페이지가 모두 비게되므로 10번 페이지의 링크를 free list 에서 삭제하고 order를 올려서 order(1)을 조사한다.
9. 이제는 비트당 매핑되는 페이지의 갯수가 2배로 늘어났으므로 비트의 인덱스도 반으로 줄어든다. 즉 index >>= 1 이 되서 인덱스가 2로 된다. 위의 인덱스 계산식을 그대로 적용해도 11 >> (1+1)=2 이므로 같다.

10. 2번 비트가 또 1이므로 10,11페이지와 버디인 8,9번 페이지가 비어있다는 것을 알 수 있다.
11. 2번 비트를 0으로 설정하고 order(1)의 free list에서 [8,9] 링크를 삭제한다.
12. 이제 우리는 [8,9,10,11] 4개의 연속된 자유 페이지를 가지고있다.
13. 이제 order(2)의 1번 비트를 보면 또 1임을 알 수 있다. 즉 [12,13,14,15] 페이지들이 모두 비어있었다는 것을 알 수있다.
14. 따라서 또 한번 병합이 일어난다.
15. 이번에는 order(2)의 free list에서 12[12,13,14,15] 링크를 삭제한다.
16. order(3)으로 올라가고 0번(1 >> 1) 비트가0임을 알 수 있다. 즉 다른 버디가 완전히 비어있지 못했다는 것을 알 수 있다. (비트가 0이라면 해제되기전에 두 버디가 모두 사용중이었고 현재 한쪽의 버디가 모두 해제되었으므로 다른 쪽 버디는 아직도 사용중이라는 것을 의미하게된다.) 따라서 우리는 병합을 할 수 없고 비트를 1로 설정하고 8[8,9,10,11,12,13,14,15]를 order(3)의 free list에 연결하게된다.




[ 그림 3-6 ]

# 4 해제 알고리즘

## 4.1 free_pages()

kernel 2.4.19 에서의 페이지 해제 소스를 분석한다. 이중에 일부분만 사용하게 된다. 특히 spin lock관련 소스는 현재는 필요가 없다. 소스를 먼저 보고 다음에 해석을 적는다.

함수 원형은 void free_pages(struct page *page, unsigned int order); 이다.

해제된 페이지는 order에 맞게 정렬되어있어야 한다. 다시 말하면 order가 1인데 3번 페이지를 해제할 수 없다는 뜻이다. order가 1이면 0,2,4,... 번호로 시작되는 2

개의 페이지를 해제하게 된다.

```
page->flags &= ~((1<<PG_referenced) | (1<<PG_dirty));
```

- 페이지의 속성을 바꾼다.

```
zone = page->zone;
```

- 페이지가 속한 zone을 알아낸다.

```
mask = (~0UL) << order;
```

- ~0은 FFFFFFFF이다. 만약 order가 3이라면 mask는 FFFFFF8이 된다. 이것을 보수를 취하면 8이 된다. 즉 mask는 페이지 번호가 order의 경계에 맞게 align이 되었는지 검사를 하는데 사용되면서 밑의 소스에서 경계값이 필요할 때도 사용한다.

```
base = zone->zone_mem_map;
```

- 현재 zone의 시작 페이지의 주소를 얻는다.

```
page_idx = page - base;
```

- 현재 해제할 페이지중에서 첫번째 페이지의 인덱스를 얻는다

```
if (page_idx & ~mask)
        BUG();
```

- 만약 페이지의 번호가 order에 맞게 정렬되지 않았으면 잘못된 연산이다. order가 3이면 mask는 FFFFFFF8이고 ~mask는 이진수로 0111이된다. page_idx가 8의 배수가 아니면 page_idx & 0111은 0이 아니고 따라서 잘못된 연산임을 알 수 있다.

```
index = page_idx >> (1 + order);
```

- 비트맵에서 몇 번 비트인가를 계산한다. 한 비트당 2^order의 크기를 가지는 페이지 블록 두개를 매핑하므로 결국 한 비트에는 2^(order+1)개의 페이지가 매핑되고 따라서 몇번 비트인지 알아내기 위해서는 페이지 번호에서 2^(order+1)을 나눠야 한다.

```
area = zone->free_area + order;
```

- free list와 비트맵에 대한 정보를 가지고있는 free_area_t 구조체에 대한 포인터를 얻는다.

zone->free_pages -= mask;

- 페이지를 해제하므로 해당 zone에서 비어있는 페이지의 갯수는 늘어나게된다. -mask는 해제될 페이지의 갯수를 나타내므로 현재 해제될 페이지가 속한 zone의 비어있는 페이지 갯수를 늘려주게된다.

while (mask + (1 << (MAX_ORDER-1))) {

- 루프 안에서 페이지를 병합하면서 order를 올려나간다. 그에 따라서 mask도 변한다. 예를 들어 처음 해제한 페이지의 order가 1이면 mask는 FFFFFFFE가 되고 MAX_ORDER는 10이므로 mask+(1<<9) != 0이다. 루프를 돌면서 order가 증가해서 order가 9까지 증가된다면 mask는 FFFFFE00이 되고 1<<9는 200이므로 mask+(1<<9)는 0이된다. 결국 order가 9가 될 때까지 계속 병합을 하도록 루프를 돈다.

```
struct page *buddy1, *buddy2;
```

- 해제할 페이지는 buddy2에 저장하고 그 짝을 buddy1에 저장한다.

```
if (!test_and_change_bit(index, area->map))
        /*
         * the buddy page is still allocated.
         */
        break;
```

- 만약 해제되는 페이지 블록의 버디 페이지가 이미 비어있는 페이지 블록이라면 해당 비트는 1일 것이다. 따라서 해당 비트가 1이라면 이제 두 버디가 모두 비어있게되므로 해당 비트를 0으로 설정해야한다. 또 해당 비트가 0이라면 두 버디가 모두 사용중이었다는 뜻이므로 비트를 1로 바꿔주어야한다.
- test_and_change_bit 함수는 해당 비트를 반전해주고 만약 해당 비트가 0일 경우에 0을 반환한다. 0을 반환하므로 if 문이 참이 되서 루프를 빠져나와서 병합이 멈추게된다.

```
/*
 * Move the buddy up one level.
 */
buddy1 = base + (page_idx ^ -mask);
buddy2 = base + page_idx;
```

- buddy1은 해제될 페이지 블록의 버디 페이지를 나타내게된다. 앞의 if문에

서 버디 페이지가 비어있는 페이지임을 알았으므로 이 buddy1은 현재 free list에 링크되어있다. 앞으로 두 버디의 병합을 위해서는 이 링크를 삭제해야한다.

- 만약 order가 0이고 해제될 페이지가 5번이라면 버디 페이지는 4번이된다. 반대로 해제될 페이지가 4번이라면 버디는 5번이 된다. 위의 코드에 대입해보면

```
buddy1 = 0 + (5 ^ 1) = 0101 ^ 0001 = 0100 = 4
```

- 5번 페이지 블록을 해제하려고할때 버디의 페이지 번호를 알아낸다는 것을 알 수 있다.

```
buddy2 = 0 + (4 ^ 1) = 0100 ^ 0001 = 0101 = 5
```

- 반대로 적용해봐도 같은 결과가 나온다.

```
memlist_del(&buddy1->list);
```

- 이제 buddy1을 free list에서 삭제한다.

```
        mask <<= 1;
        area++;
        index >>= 1;
        page_idx &= mask;
    }
```

- 이제 order를 증가시킨다. 다음 order로 넘어가기 위해 mask는 한 비트 왼쪽으로 쉬프트하게되고 area도 증가시킨다. index는 order가 증가하므로 2로 나누어준다. 페이지 번호도 2로 나눈 몫으로 계산하게된다. order가 1 증가하는 것은 다루는 페이지 블록의 크기가 2배가 된다는 것이므로 당연히 2로 나눠주는 것이다.

```
memlist_add_head(&(base + page_idx)->list, &area->free_list);
```

- 더 이상 병합을 할 수 없는 최종적으로 남은 페이지 블록은 현재 order의 free list에 추가시켜준다.

## 5 할당 알고리즘

gfp_mask는 여유 페이지 프레임을 어떻게 찾을 것인지 지정하며, 다음 플래그로

구성된다.

_ _GFP_WAIT

The kernel is allowed to block the current process waiting for free page frames.

_ _GFP_HIGH

The kernel is allowed to access the pool of free page frames left for recovering from very low
memory conditions.

_ _GFP_IO

The kernel is allowed to perform I/O transfers on low memory pages in order to free page frames.

_ _GFP_HIGHIO

The kernel is allowed to perform I/O transfers on high memory pages in order free page frames.

_ _GFP_FS

The kernel is allowed to perform low-level VFS operations.

_ _GFP_DMA

The requested page frames must be included in the ZONE_DMA zone

_ _GFP_HIGHMEM

The requested page frames can be included in the ZONE_HIGHMEM zone.

리눅스에서는 주로 다음처럼 각 플래그의 조합을 사용한다.

Table 7-5. Groups of flag values used to request page frames

| Group name | Corresponding flags |
|---|---|
| GFP_ATOMIC | __GFP_HIGH |
| GFP_NOIO | __GFP_HIGH __GFP_WAIT |
| GFP_NOHIGHIO | __GFP_HIGH __GFP_WAIT __GFP_IO |
| GFP_NOFS | __GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO |
| GFP_KERNEL | __GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS |
| GFP_NFS | __GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS |
| GFP_KSWAPD | __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS |
| GFP_USER | __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS |
| GFP_HIGHUSER | __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS __GFP_HIGHMEM |

[ 표 5-1 ]

zonelist 는 zone 디스크립터의 배열을 가르키는데 비어있는 페이지를 찾기위해 검색할 zone을 가르킨다.

Table 7-6. Zone modifier lists

| __GFP_DMA | __GFP_HIGHMEM | Zone list |
|---|---|---|
| 0 | 0 | ZONE_NORMAL + ZONE_DMA |
| 0 | 1 | ZONE_HIGHMEM + ZONE_NORMAL + ZONE_DMA |
| 1 | 0 | ZONE_DMA |
| 1 | 1 | ZONE_DMA |

[ 표 5-2 ]

## 5.1 __alloc_pages

```
 /*
 * This is the 'heart' of the zoned buddy allocator:
 */
struct page * __alloc_pages(unsigned int gfp_mask, unsigned int order, zonelist_t
*zonelist)
{
```

- 요청한 페이지를 위한 속성과 페이지의 갯수, 페이지를 검색할 zone을 인자로 보낸다. 버디 할당의 심장부라고 주석이 써져있다.

```
    unsigned long min;
    zone_t **zone, * classzone;
    struct page * page;
    int freed;
```

- min은 zone에서 페이지를 할당한 후에 zone에 남겨져야할 최소한의 페이지 갯수를 말한다.
- classzone은 우선적으로 페이지를 찾아볼 zone을 말한다.

```
    zone = zonelist->zones;
    classzone = *zone;
```

- zonelist에서 첫번째 zone을 찾아서 classzone 변수에 저장한다. 페이지를 찾을 때는 항상 classzone에 저장된 zone부터 검색을 하게된다.

```
    min = 1UL << order;
```

- min을 요청한 페이지의 갯수로 맞춘다.

```
    for (;;) {
            zone_t *z = *(zone++);
            if (!z)
                    break;
```

- zonelist에 있는 모든 zone을 검색하면서 비어있는 페이지를 찾는다.
- 모든 zone을 검색해도 빈 페이지를 찾지 못하면 루프를 끝낸다.

```
            min += z->pages_low;
```

- pages_low는 zone의 페이지 균형 알고리즘에서 사용할 낮은 threshold를 말한다. z에 pages_low와 요청된 페이지의 수를 합한 것 이상으로 비어있는 페이지가 있어야 하기때문에 min에 이 값을 더한다.

```
        if (z->free_pages > min) {
                page = rmqueue(z, order);
                if (page)
                        return page;
        }
}
```

- zone에 충분하게 빈 페이지들이 있으면 rmqueue 함수를 호출해서 빈 페이지의 링크를 삭제해서 반환한다.

```
classzone->need_balance = 1;
mb();
if (waitqueue_active(&kswapd_wait))
        wake_up_interruptible(&kswapd_wait);
```

- pages_low 만큼 빈 페이지가 남아야하는데 그렇지 못한 상황이라면 페이지 균형을 맞추기 위한 준비를 한다. 또 kswapd_wait 프로세스를 깨워서 이 zone에서 빈 페이지들을 만들기 시작한다.

```
zone = zonelist->zones;
min = 1UL << order;
```

- 아까와 같이 다시 페이지를 할당하기위해 변수를 설정한다.

```
for (;;) {
        unsigned long local_min;
        zone_t *z = *(zone++);
        if (!z)
                break;

        local_min = z->pages_min;
```

- 이번에는 pages_low가 아니라 pages_min으로 계산한다. pages_min은 zone에 남아있어야할 최소한의 빈 페이지 갯수를 말한다.
- 지금은 kswapd_wait가 작동하여 어느정도 빈 페이지를 만들어줄 것으로 기대하므로 최소한의 갯수만 남도록 하게되는 것이다.

```
        if (!(gfp_mask & __GFP_WAIT))
                local_min >>= 2;
```

- 만약 페이지를 요청한 프로세스가 빈 페이지를 기다릴 수 없는 상황이라면 zone에 여유를 줄여서 할당을 시도하게된다.

```
        min += local_min;
        if (z->free_pages > min) {
                page = rmqueue(z, order);
                if (page)
                        return page;
        }
}
```

- 위에서와 마찬가지로 zone에서 빈 페이지의 갯수가 최소한의 갯수를 넘는다면 rmqueue 함수를 호출해서 페이지를 반환한다.

## 5.2 rmqueue()

alloc_pages 함수는 적합한 여유 페이지 프레임 수를 가진 zone을 발견하면 rmqueue() 함수를 호출하여 해당 zone에서 블록을 할당한다. 이 함수는 두개의 인자를 가지는데 하나는 zone의 디스크립터의 포인터이고 하나는 요청한 페이지 블록의 크기이다. 페이지 할당이 성공하면 할당한 첫번째 페이지 프레임의 페이지 디스크립터의 주소를 반환한다. 그러면 alloc_pages()는 그 주소를 그대로 반환한다. 실패하면 NULL을 반환하고 alloc_pages()는 다음 zone을 검색하게된다. 이 함수는 페이지 요청에 맞게 order를 찾는 일을 한다. 예를 들어 페이지 한개를 할당받기 위해 order=0으로 호출했어도 free list를 검색해서 빈 페이지가 없으면 order=1을 검색해서 빈 페이지를 찾아서 반으로 나누고 반은 반환하고 반은 order=0의 free list에 연결한다.

```
static struct page * rmqueue(zone_t *zone, unsigned int order)
{
```

- 메모리를 얻을 zone과 페이지 개수를 인자로 받는다.

```
        free_area_t * area = zone->free_area + order;
        unsigned int curr_order = order;
```

- free list와 비트맵의 포인터를 저장한다.

```
        struct list_head *head, *curr;
```

```
unsigned long flags;
struct page *page;

spin_lock_irqsave(&zone->lock, flags);
do {
        head = &area->free_list;
        curr = head->next;
```

- free list에 연결된 첫번째 페이지 블록

```
        if (curr != head) {
```

- free list가 비어있지 않으면

```
                unsigned int index;

                page = list_entry(curr, struct page, list);
```

- 첫번째 페이지 블록에서 첫번째 페이지의 디스크립터를 얻는다.

```
                if (BAD_RANGE(zone,page))
                        BUG();
                list_del(curr);
```

- free list에서 얻어낸 페이지 블록을 리스트에서 삭제한다.

```
                index = page - zone->zone_mem_map;
```

- 페이지 디스크립터 테이블에서 인덱스를 계산한다.

```
                if (curr_order != MAX_ORDER-1)
                        MARK_USED(index, curr_order, area);
```

- 만약 현재 order가 9이면 해당 페이지를 매핑하는 비트를 토글하지 않는다. 결국 order가 9일때 비트맵을 사용하지 않는 것이 된다. order 9의 비트맵은 항상 모두 0이다. 왜냐면 비트맵은 버디 페이지의 병합을 위해서 사용하는 것인데 2^9 페이지 블록은 최대 버디 페이지 블록의 크기가 2^9이므로 병합할 수 없기 때문이다. => 매우 중요하다.

```
                zone->free_pages -= 1UL << order;
```

- zone 안에 있는 빈 페이지들의 개수를 다시 계산한다. (1UL << order) = 2^order 이다.

```
                        page = expand(zone, page, index, order, curr_order,
area);
```
- expand 함수는 다음 절에서 따로 설명한다.

```
                        set_page_count(page, 1);
```
- 페이지 카운트를 1로 설정한다.

```
                        return page;
```
- 페이지 디스크립터를 반환한다.

```
                }
                curr_order++;
                area++;
```
- 위에서 free list가 비어있다면 다음 order로 올라가서 찾아본다.

```
        } while (curr_order < MAX_ORDER);
```
- curr_order가 9가 될때까지 시도한다.

```
        return NULL;
}
```
- 실패하면 NULL을 반환한다.

## 5.3 expand()

이 함수는 높은 order의 페이지 블록을 나눠서 사용자가 요구한 크기의 페이지 블록을 반환하고 남은 페이지 블록들을 각 order에 맞는 free list로 연결하고 해당 order의 비트맵을 처리하는 일을 한다. 예를 들어 order가 1인 페이지를 요청했는데 order 3에 비어있는 페이지 블록이 있다면 order 3의 페이지 블록을 둘로 나눠서 하나는 order 2의 free list에 연결하고 하나는 다시 둘로 나눠서 하나는 반환하고 하나는 order 1의 free list에 연결한다.

```
static inline struct page * expand (zone_t *zone, struct page *page,
         unsigned long index, int low, int high, free_area_t * area)
{
```
- high는 비어있는 페이지 블록이 있는 높은 order이고 low는 요청한 페이지

블록의 order이다.

unsigned long size = 1 << high;

- 현재 처리하고있는 페이지 블록의 크기는 2^high이다.

while (high > low) {

- 만약 high와 low가 같다면 page를 반환하고 종료한다.

area--;
high--;
size >>= 1;

- 루프를 돌때마다 order를 1 감소한다.

list_add(&(page)->list, &(area)->free_list);

- 인덱스가 앞에있는 페이지 블록은 free list에 연결하고 뒤에 있는 페이지 블록은 루프를 돌면서 반으로 나눈다. 계속 나누다가 요청한 크기의 페이지 블록이 되면 페이지 디스크립터를 반환하게된다.

MARK_USED(index, high, area);

- 해당 비트맵의 비트를 반전시킨다.

index += size;
page += size;

- 위에서는 페이지 블록을 나눠서 앞에있는 버디는 free list에 연결하고 뒤에 있는 버디는 남겼었다. 남은 버디를 다루기 위해서 index와 페이지 디스크립터를 남은 버디에 맞게 조정준다.

}
return page;
}

- 뒤에 남은 버디에서 첫번째 페이지의 페이지 디스크립터를 반환한다.

# Bibliography

[1] Abhishek Nayani & Mel Gorman & Rodrigo S. de Castro. Memory Management in Linux: Desktop Companion to the Linux Source Code

[2] Daniel P. Bovet & Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001, ISBN 81-7366-233-9.

[3] Intel Architecture. *Intel Pentium III Processor Manuals*. http://www.intel.com/design/PentiumIII/manuals/index.htm.

[4] Martin Devera. *Functional Callgraph of the Linux VM*. http://luxik.cdi.cz/~devik/mm.htm.
Contains a patch for gcc which was used to create the call-graph poster provided with this doc.

[5] Je® Bonwick. *The Slab Allocator: An Object Caching KernelMemory Allocator*. http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html.
This paper presents a comprehensive design overview of the SunOS 5.4 kernel memory allocator.