가상메모리 관리 기법

Task_struct

- 가상메모리 관련 부분도 여기에 저장
- Mm이라는 필드에 저장되어 있음
- 얘는 mm_struct를 가리킴

```
struct kioctx_table;
struct mm_struct {
        struct {
                struct vm_area_struct *mmap;            /* list of VMAs */
                struct rb_root mm_rb;
                u64 vmacache_seqnum;                   /* per-thread vmacache */
#ifdef CONFIG_MMU
                unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long len,
                                unsigned long pgoff, unsigned long flags);
#endif
                unsigned long mmap_base;        /* base of mmap area */
                unsigned long mmap_legacy_base; /* base of mmap area in bottom-up allocations */
#ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES
                /* Base adresses for compatible mmap() */
                unsigned long mmap_compat_base;
                unsigned long mmap_compat_legacy_base;
#endif
                unsigned long task_size;        /* size of task vm space */
                unsigned long highest_vm_end;   /* highest vma end address */
                pgd_t * pgd;

#ifdef CONFIG_MEMBARRIER
                /**
                 * @membarrier_state: Flags controlling membarrier behavior.
                 *
                 * This field is close to @pgd to hopefully fit in the same
                 * cache-line, which needs to be touched by switch_mm().
                 */
```

https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h

Mm_struct

- 크게 3개로 나눌 수 있음
  - vm_area_struct
  - Pgd
  - Start_code, start_data, start_data

vm_area_struct

- 커널에서 같은 속성을 가진 연속된 영역을 regio이라 부름
- 이런 각각의 region을 vm_area_struct로 관리함
- vm_area_struct 은 rbtree로 관리하는데, mm_rb와 mmap_cache 등이 더 있음

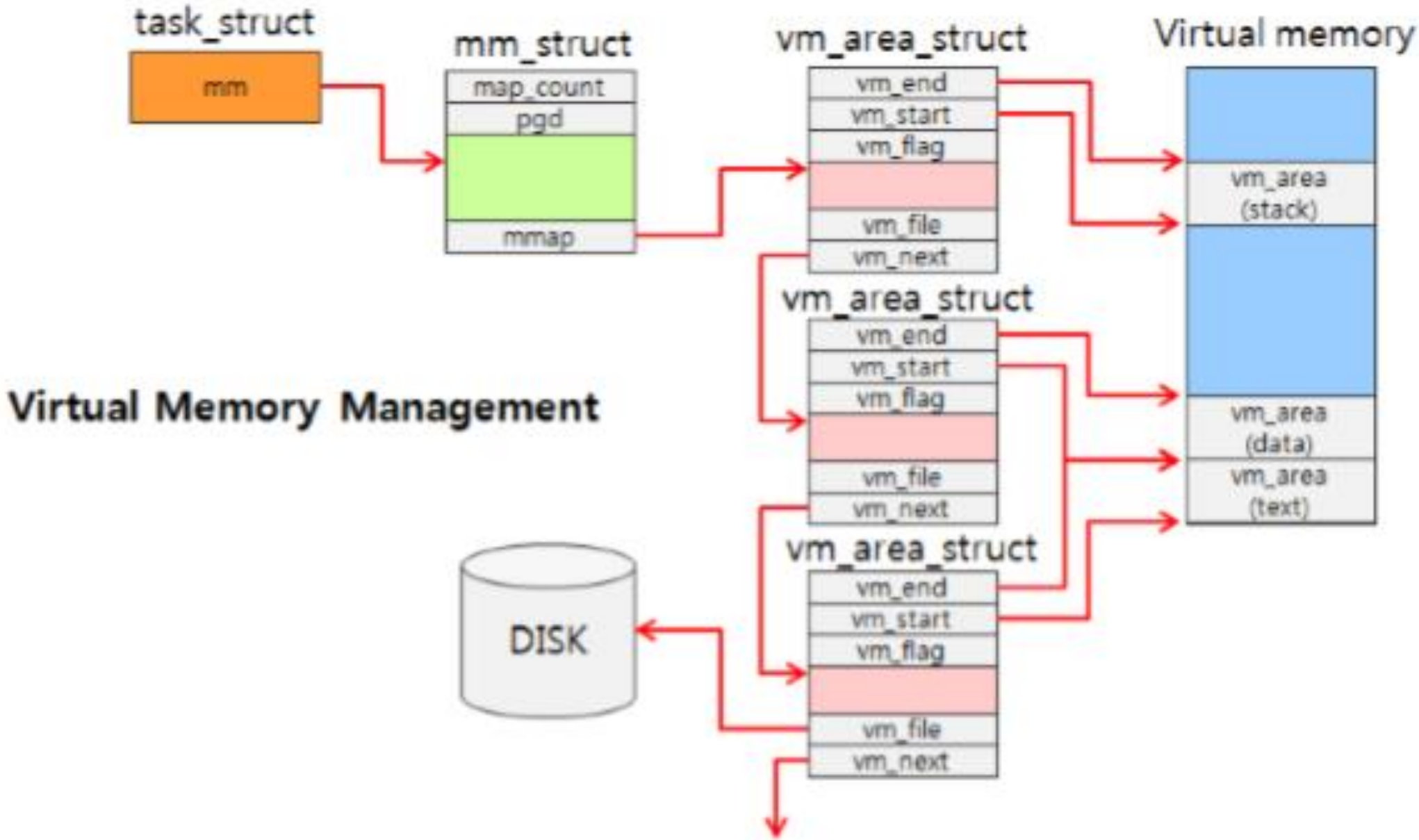
pgd

- Page directory 시작 주소를 여기에 저장함

Start_code, start_data, start_data

- 이름 그대로 code영역의 시작주소, data영역의 시작주소 등을 저장함

- 공통된 속성 (.text, .data, …)을 가진 page들이 vm_area를 구성
- vm_area를 vm_area_struct로 관리
- 이러한 같은 task_struct를 가진 vm_area_struct 들이 mm_struct 내에 관리



Virtual Memory Management

가상메모리 할당 / 해제

- vm_area_struct 의 할당 / 해제
- Page의 할당 / 해제

vm_area_struct 의 할당 / 해제

- 새로 할당할 가상 공간을 찾는다
  - Arch_get_unmmaped_area() 함수
- 해당하는 새로운 vm_area_struct 할당
- vm_area_struct가 인접한 vm_area_struct과 속성이 같다면 병합
  - do_mmap_pgoff()

가상 / 물리 메모리 연결 및 변환

ELF파일의 프로그램 수행 과정(메모리 관점)

- 태스크 생성
- 태스크에 가상 주소 공간 제공
  - 필요시 물리 메모리 일부 할당
- 태스크가 원하는 디스크상 내용을 물리 메모리에 올림
- 이 물리 메모리의 실제 주소와 태스크의 가상 주소 공간을 연결

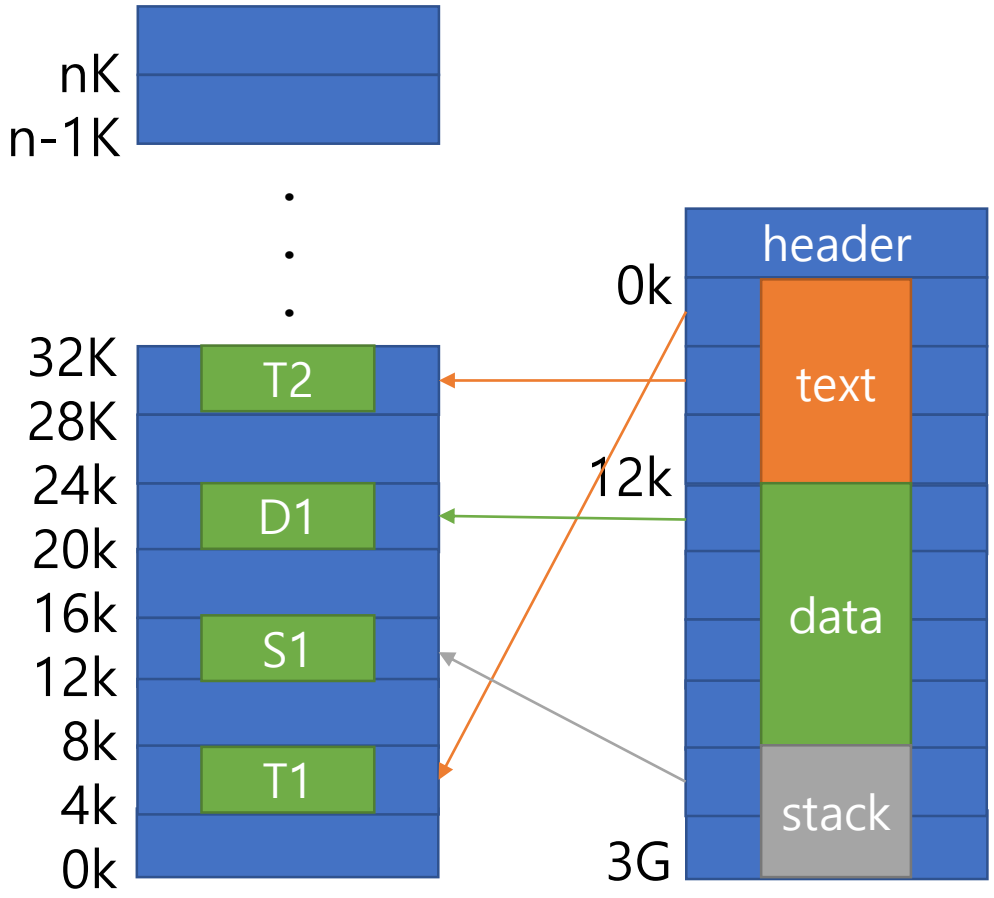
sys_execve() 호출 과정

- 프로그램 가정: ELF헤더, phdr(프로그램 헤더), 3개의 section(텍스트 12kb, 데이터 16kb, 스택 8kb)
- 텍스트의 각 페이지를 t1, t2, t3로 명명
- 데이터의 각 페이지를 d1, d2, d3, d4로 명명
- 스택의 각 페이지를 s1, s2로 명명

- free한 페이지 프레임들을 할당

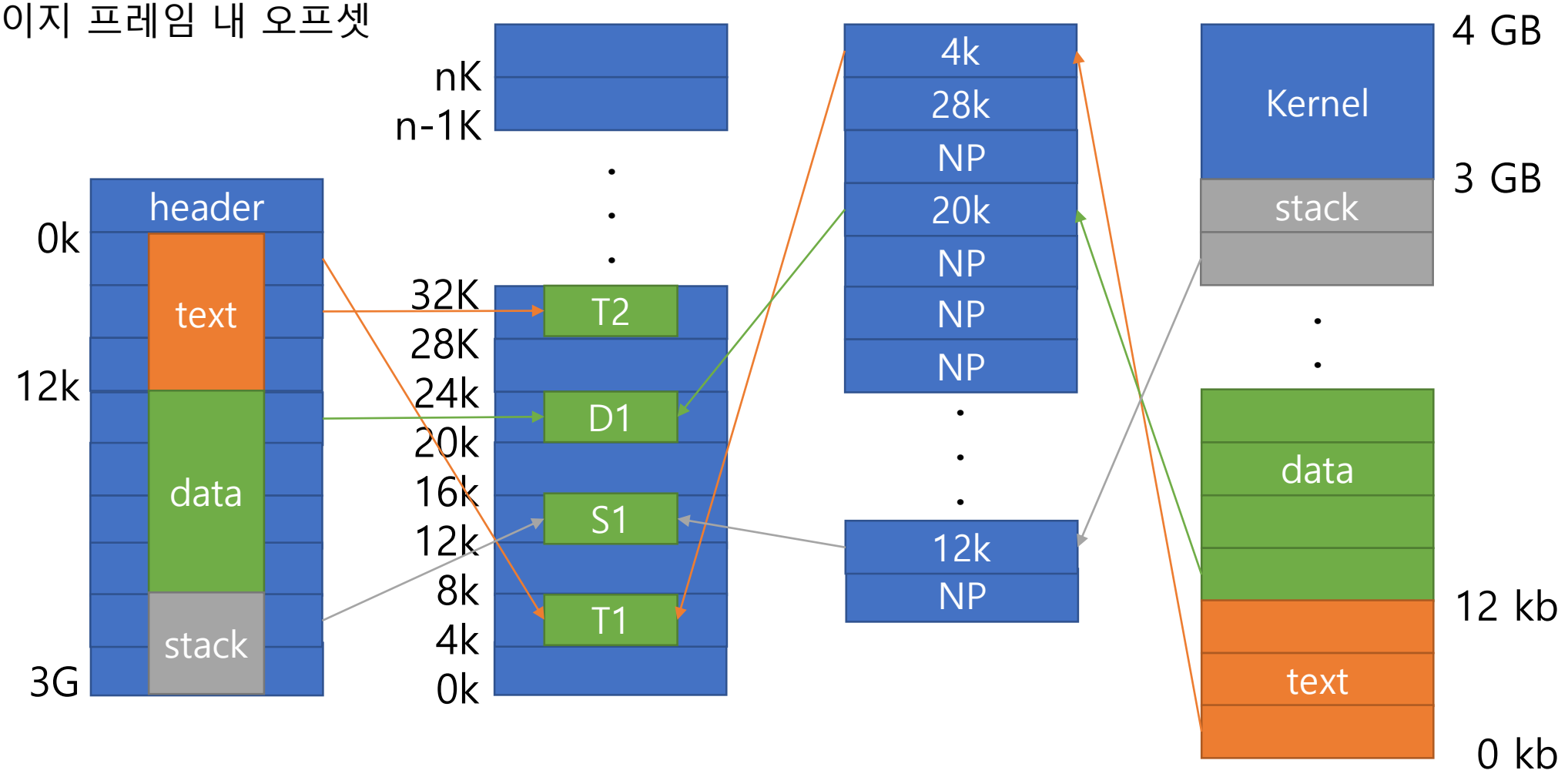
- 파일 시스템에게 파일 내용 일부를 요청

- 읽혀진 내용을 할당 받은 페이지 프레임에 적재
  (가정: t1을 페이지 프레임 4kb, t2를 페이지 프레임 28kb
  d1을 페이지 프레임 20kb, s1을 12kb 위치에 적재)



Page Table

- 페이지 크기는 4kb로 가정
- 가상 주소를 물리 주소로 변환하는 주소 변환 정보를 기록한 테이블
- 가상 주소를 페이지 크기로 나눔
- 몫은 테이블의 엔트리를 탐색하는 인덱스로 사용
- 나머지는 페이지 프레임 내 오프셋



Page Table 계산

- CPU가 가상 주소 1000에 접근하려 가정(즉, 가상 메모리의 t1 페이지)
- 몫: 0 / 나머지: 1000
- 따라서, 인덱스 0 페이지 테이블 탐색 -> 페이지 테이블 첫번째 엔트리 4k 발견
- 즉, 이 페이지는 4kb로 시작하는 페이지 프레임에 존재
- 따라서 물리주소는 4kb + 1000 byte = 5096 byte

Quiz: 10000번지를 접근해 보자

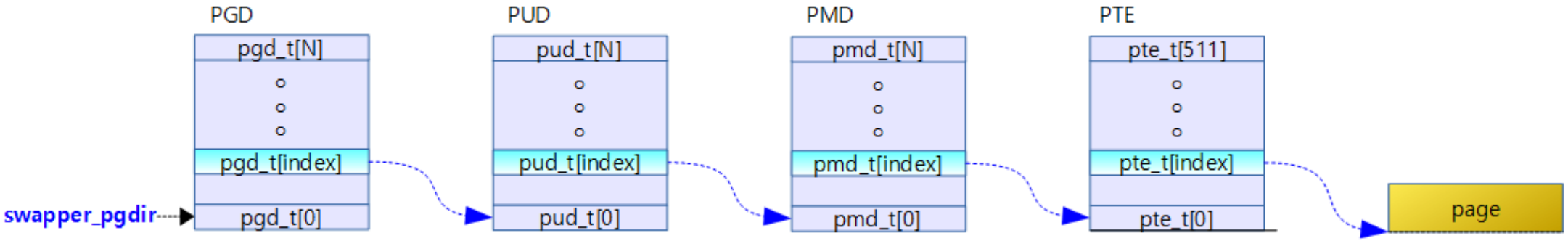
Solution
10000 / 4096 = 2
10000 % 4096 = 1808
Result: NP + 1808
-> Page Fault 발생 -> Trap -> IDT 테이블 거침 -> 페이지 폴트 핸들러(6장 참고)

Demand Paging
- Free한 프레임을 할당
- 필요한 페이지를 이 페이지 프레임에 읽어다 놓음
- 적재하면서 페이지 테이블에 적재된 페이지 프레임 번호를 기록
- 주소 변환 과정을 다시 수행

리눅스커널의 4단계 페이징

- 각 태스크마다 하나의 페이지 테이블을 사용하면 페이지 테이블의 엔트리 개수가 너무 많아져서 제작(용량을 너무 많이 씀)

- task_struct에는 mm이라는 필드가 존재
- mm필드는 mm_struct라는 자료구조를 가리킴
- mm_struct의 pgd(Page Global Directory)는 PGD의 페이지 프레임 번호를 가짐
- 이를 통해 가상 주소 일부를 이용하여 인덱싱하면 PMD(Page Middle Directory)주소를 얻음
- 다시 가상 주소 일부를 이용하여 인덱싱하면 PTE(Page Table Entry)를 얻음
- PTE에서 다시 가상 주소 일부를 이용하면 실제 접근할 페이지 프레임 주소를 얻음



```
static struct page * follow_page_pte(struct vm_area_struct *vma, unsigned long address,
                                     unsigned int flags, unsigned int *page_mask)
{
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmt;
        pte_t *ptep, tpe;

        pgd = pgd_offset(mm, address);
        if( pge_none(*pgd) || pgd_bad(*pgd))            goto out;
        pud = put_offset(pgd, address);
        if(pud_none(*pud))                              goto out;
        pmd = pmd_offset(pud, address);
        if(pmd_none(*pmd) || pmd_bad(*pmd))             goto out;

        ...

        return follow_page_pte(vma, address, pmd, flags);
}
```

MMU(Memory Management Unit)

- CPU가 메모리에 접근하는 것을 관리하는 컴퓨터 하드웨어 부품
- Intel 32bit CPU -> 2단계의 페이지 테이블 지원
- 알파 CPU -> 3단계 페이지 테이블 지원
- 64bit CPU지원을 위해 커널 2.6.11부터 4단계 페이징을 지원

Intel 32bit CPU MMU

- page directory를 cr3 레지스터에 저장
- 가상 주소 상위 10비트를 이용하여 page directory에서 오프셋으로 사용
- 이 결과로 특정 entry로 가는데, 이 entry에는 page table 시작 주소가 저장되어 있음
- 따라서, 네 단계의 페이지 테이블 중 PMD, PUD가 무시됨
- 현재 실행중인 태스크의 pgd값이 cr3 레지스터에 저장됨

vmalloc과 vfree

- 물리 메모리 상 연속인 메모리 공간이 넉넉하지 않아서 사용
- 페이지 테이블로 가상과 물리 메모리를 연결 가능
- 가상 메모리만 연속
- 물리 메모리에서는 연속이 아니어도 됨

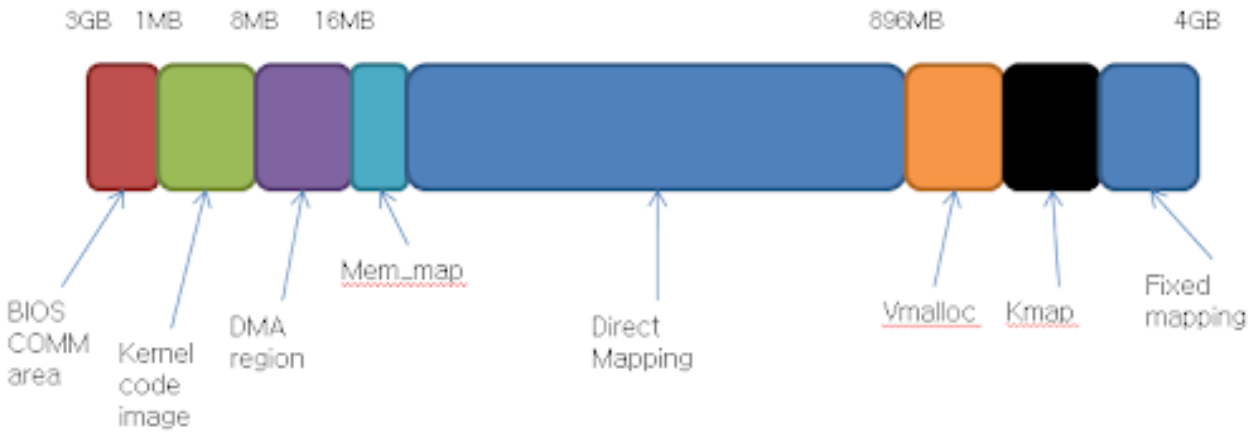
가상메모리 단점

- 물리 메모리 변환 과정 필요
- 이 과정으로 프로그램 수행 시간을 지연 시킬 가능성이 있음
- 메모리 접근 시간에 대한 예측성을 떨어뜨림

위와 같은 단점을 해결하기 위해 하드웨어(Hardware Address Translation, MMU)를 사용
또는, TLB같은 페이지 테이블 엔트리 캐시를 이용

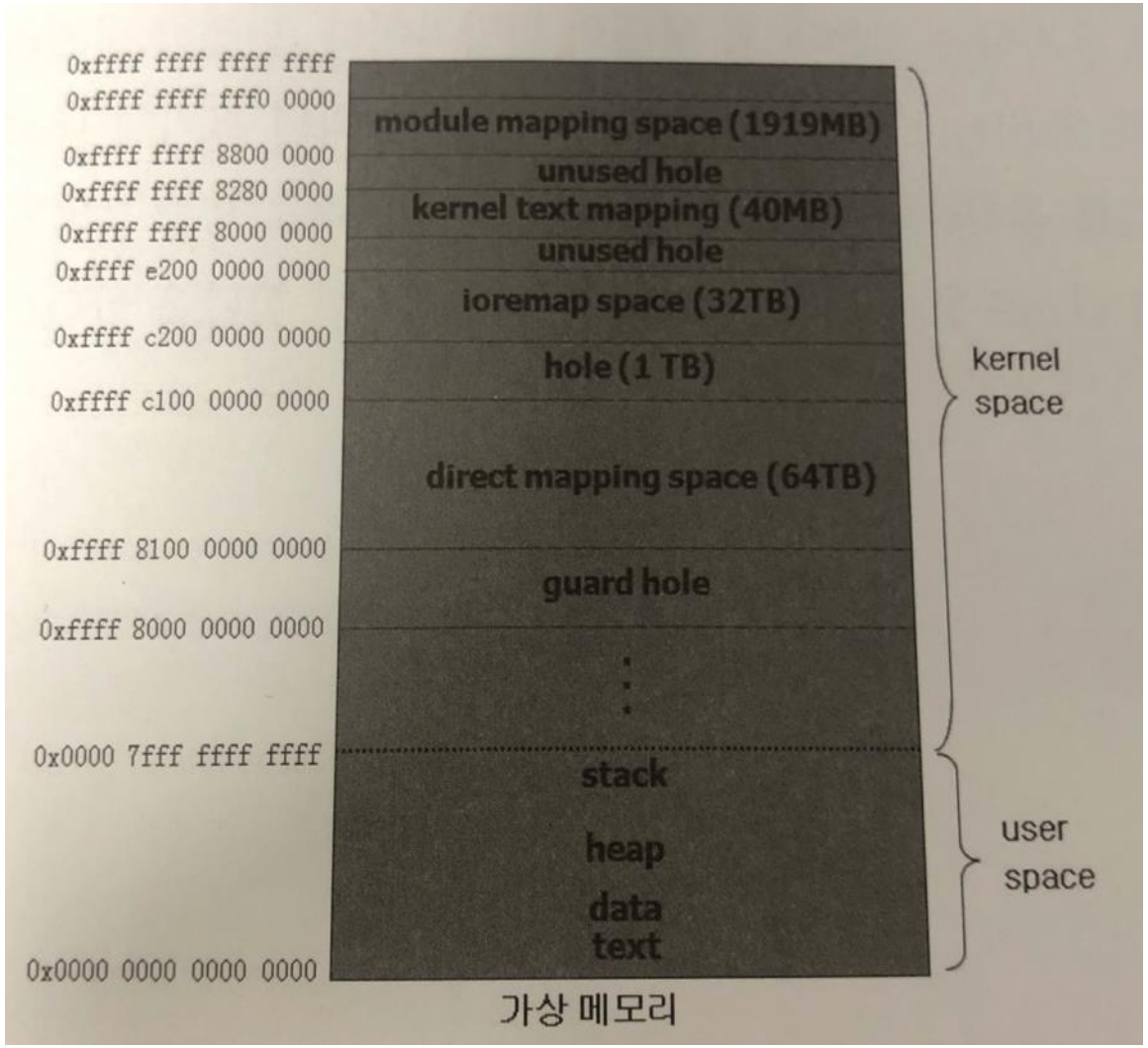
커널 주소 공간

32bit CPU Kernel Address Space



- 3GB – 896MB까지는 ZONE_NORMAL 영역
- 16MB까지 ZONE_DMA 영역
- Mem_map은 물리메모리를 표현하기 위한 페이지 프레임 자료구조가 들어있는 배열영역
- Direct Mapping 영역은 동적 할당 공간으로 사용됨 가상주소와 물리주소는 이영역으로 연결되어있음
- 896MB이후 부분은 임시 매핑하여 사용하기 위한 공간으로 씀 (ZONE_HIGHMEM)

64bit CPU Kernel Address Space



- 64bit는 가상주소가 0x7f~~ff~0xf~~ff (128tb-16eb)
- 32bit와 달리 Direct Mapping가 64TB로 엄청 크게할당
- 너무 공간이 남아서 0x7f~~ff-0xffff80~00는 비사용

Slub, Slob

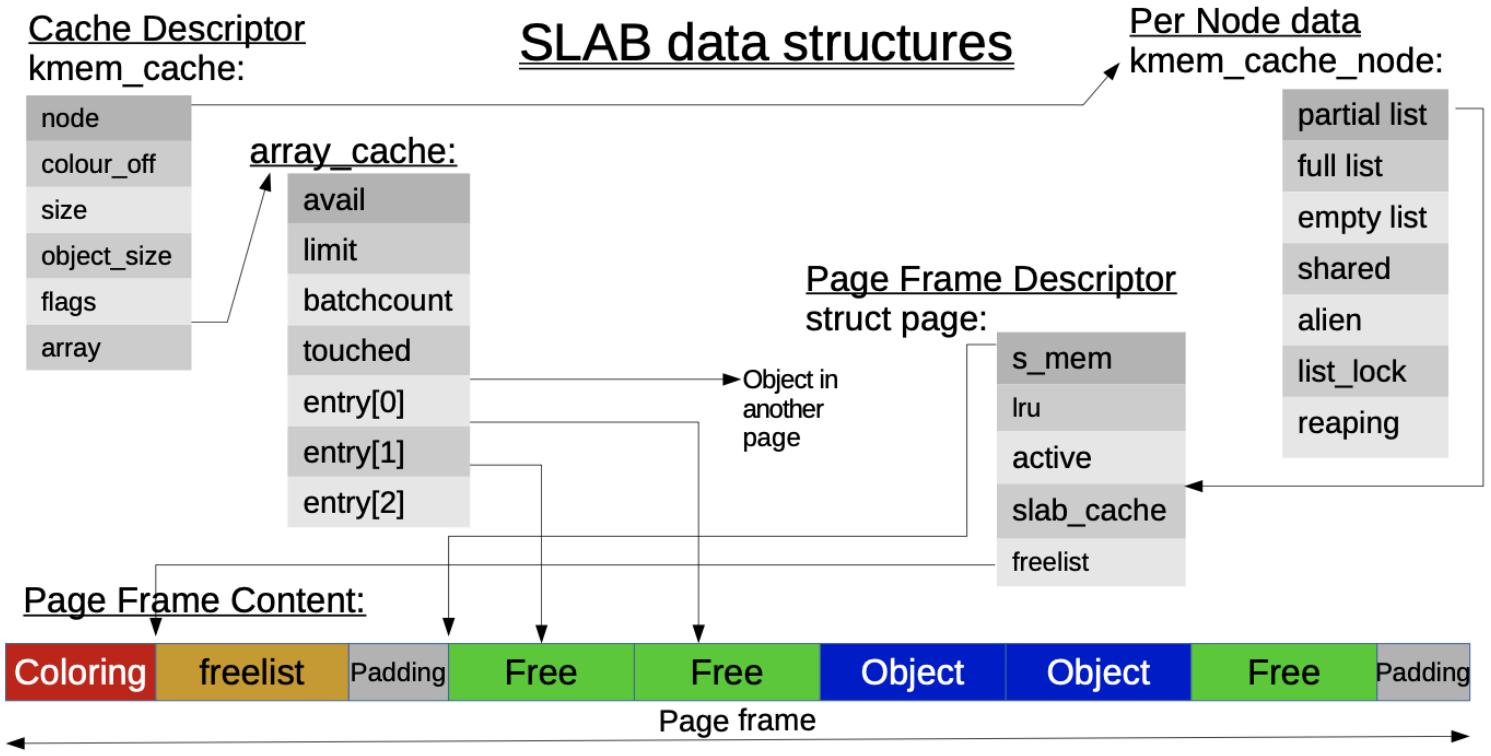
Slab 한계

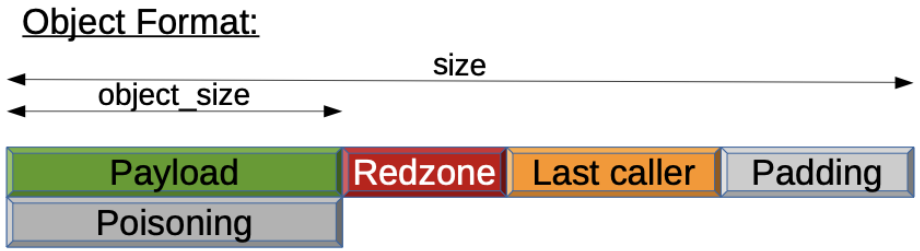
1. object 큐 오버헤드
   CPU, node 각각에 따른 Slab
   CPU마다 array_cache에 포인터 배열

2. partial 리스트 오버헤드
   Slab – 각 노드의 partial 리스트
   Slub – global 리스트

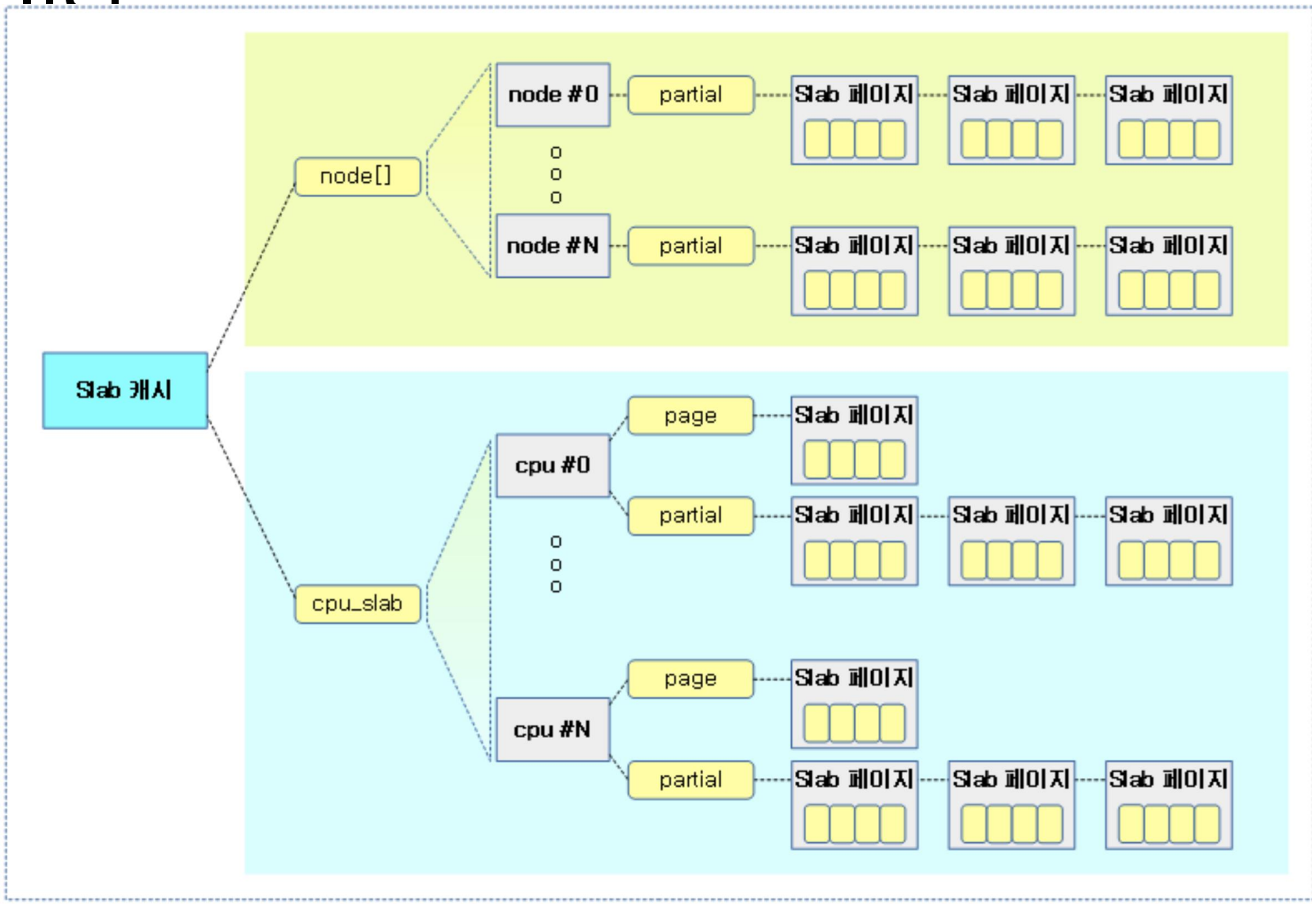
3. Slab 자체 오버헤드
   Slab 관리를 위한 메타 데이터

Slab의 한계 - object 큐 오버헤드

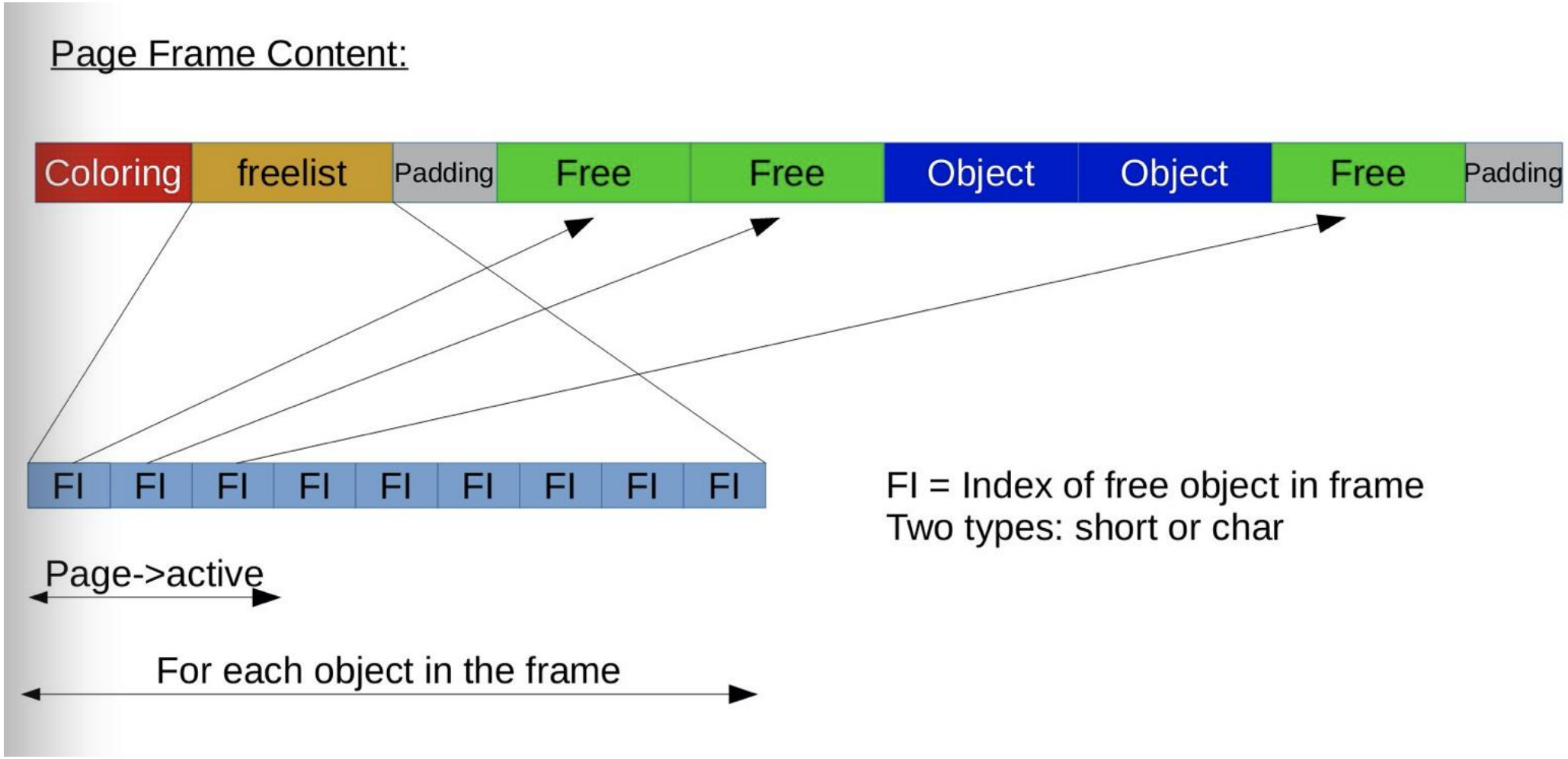




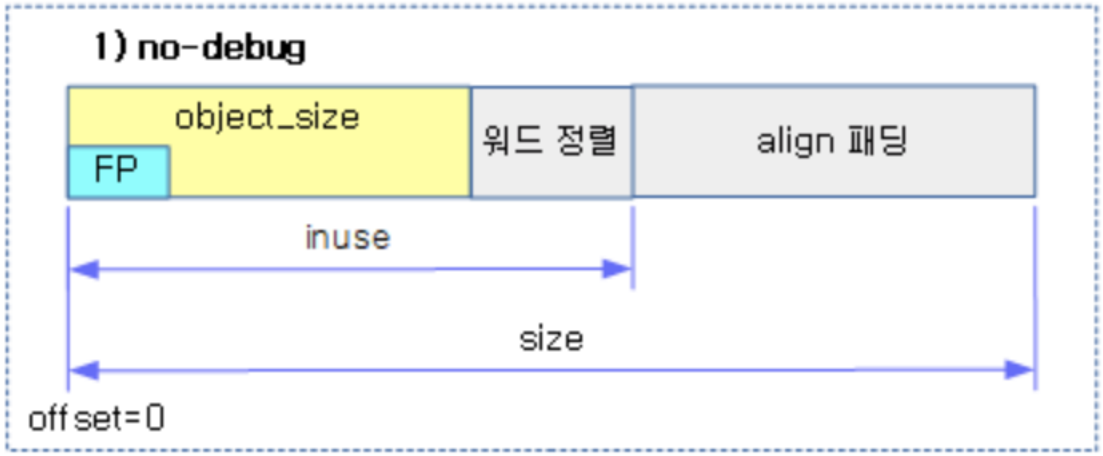
Slab의 한계 - 객체 큐 오버헤드& partial list



Slab의 한계 - Slab 자체 오버헤드



1) 메타 정보 없는 slub object



Slub vs slab 차이점

Page struct 변화

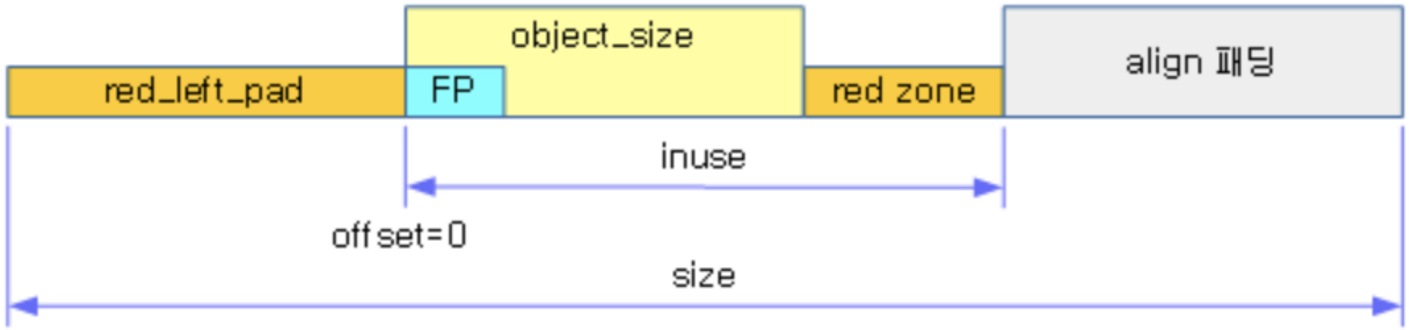
1. void *freelist;
   첫번째 free된 object 포인터
2. short unsigned int inuse;
   메타 데이터로 인해 추가된 공간을 제외한 실제 크기
3. short unsigned int offset;

통합 여부

Slub : 비슷한 slab끼리 서로 혼합해서 사용 -> 메모리 효율 증가

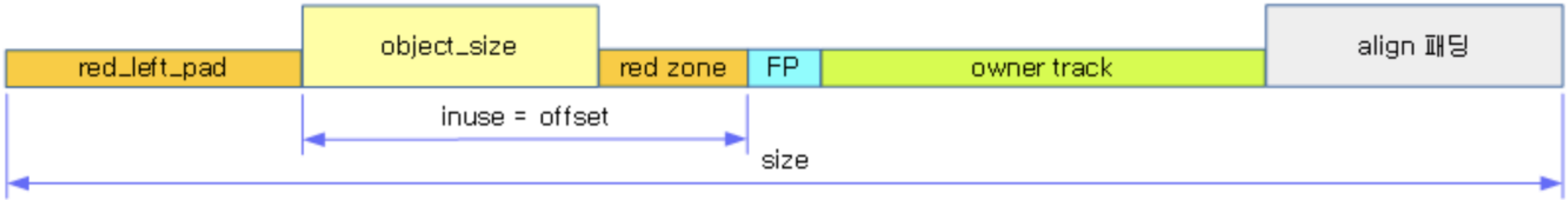
Global 리스트

2) red-zone

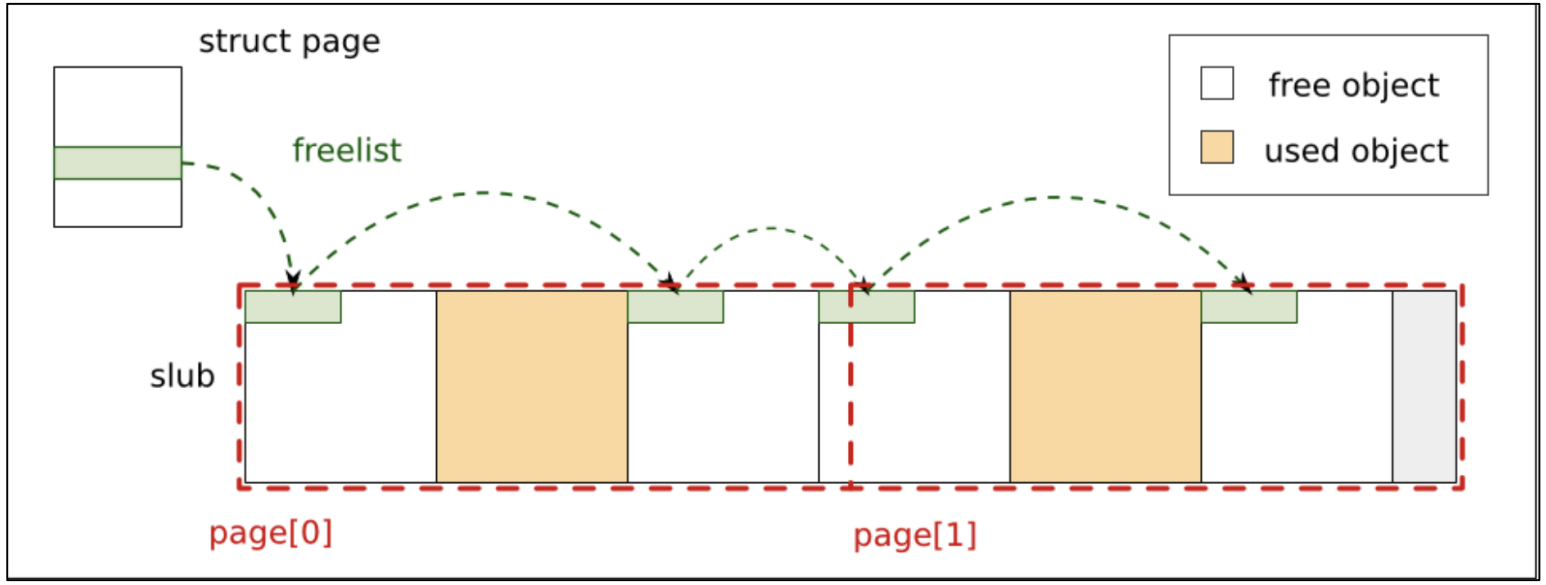


3) poison, rcu, ctor

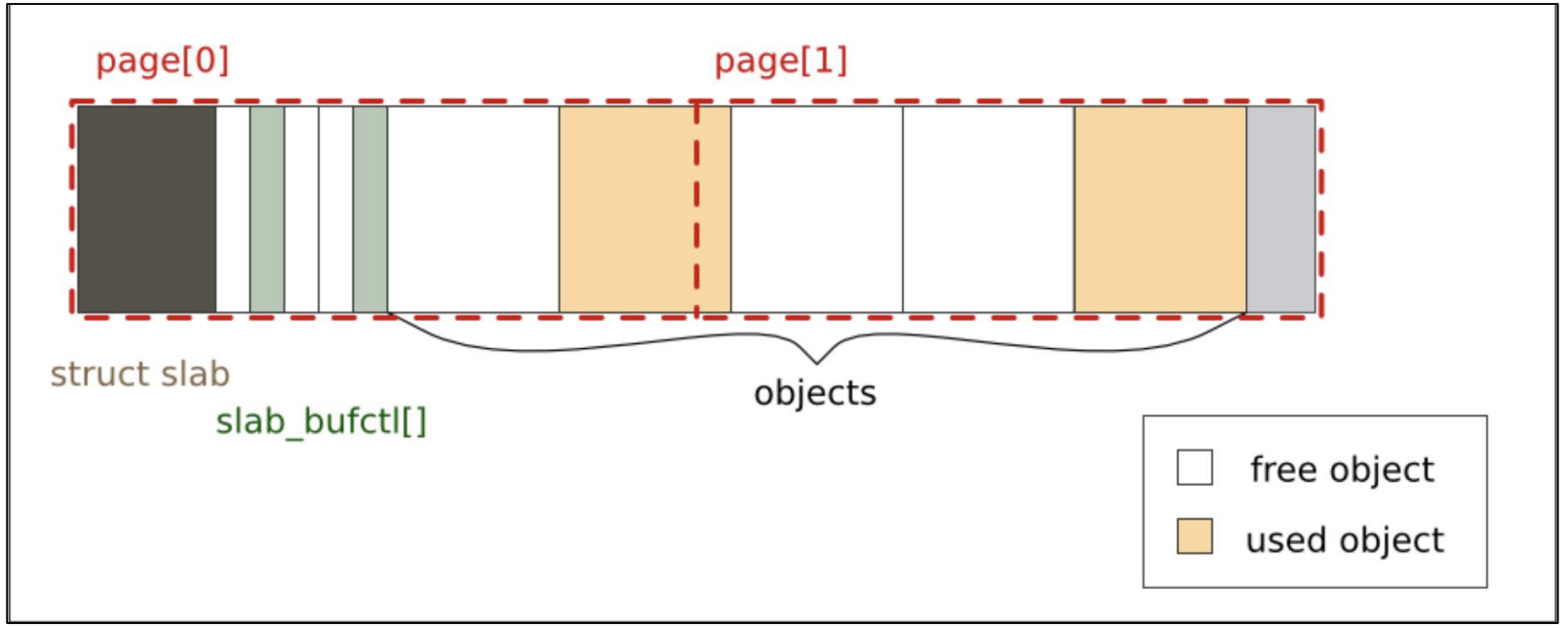
5) red-zone + poison + owner track



Slub


struct page
freelist
free object
used object
slub
page[0]
page[1]

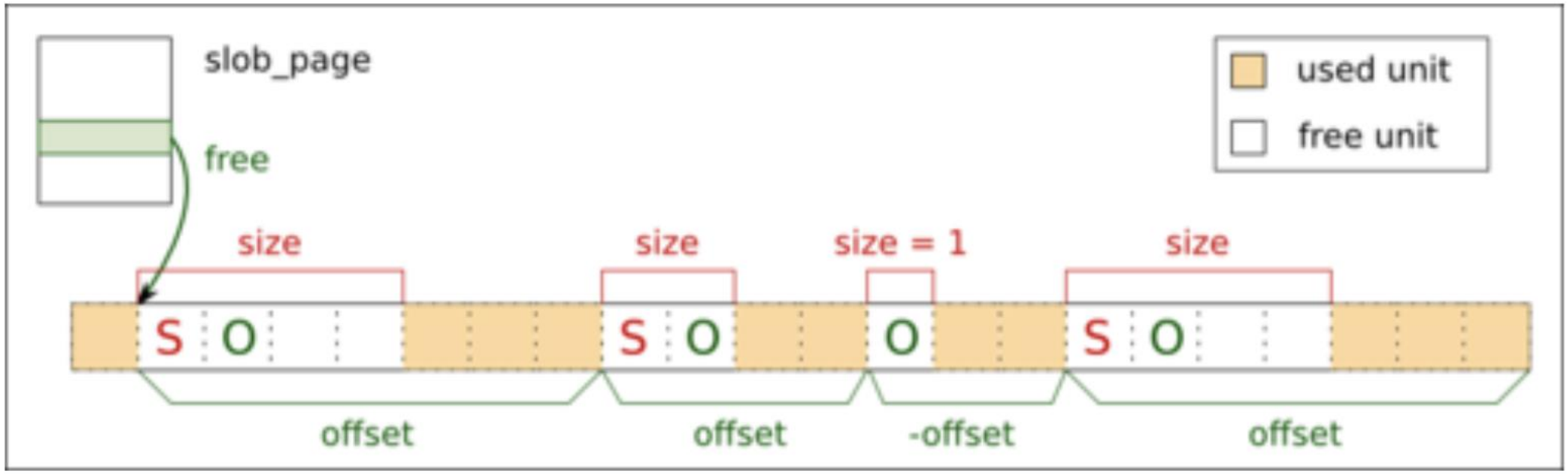
Slab


page[0]
page[1]
struct slab
slab_bufctl[]
objects
free object
used object

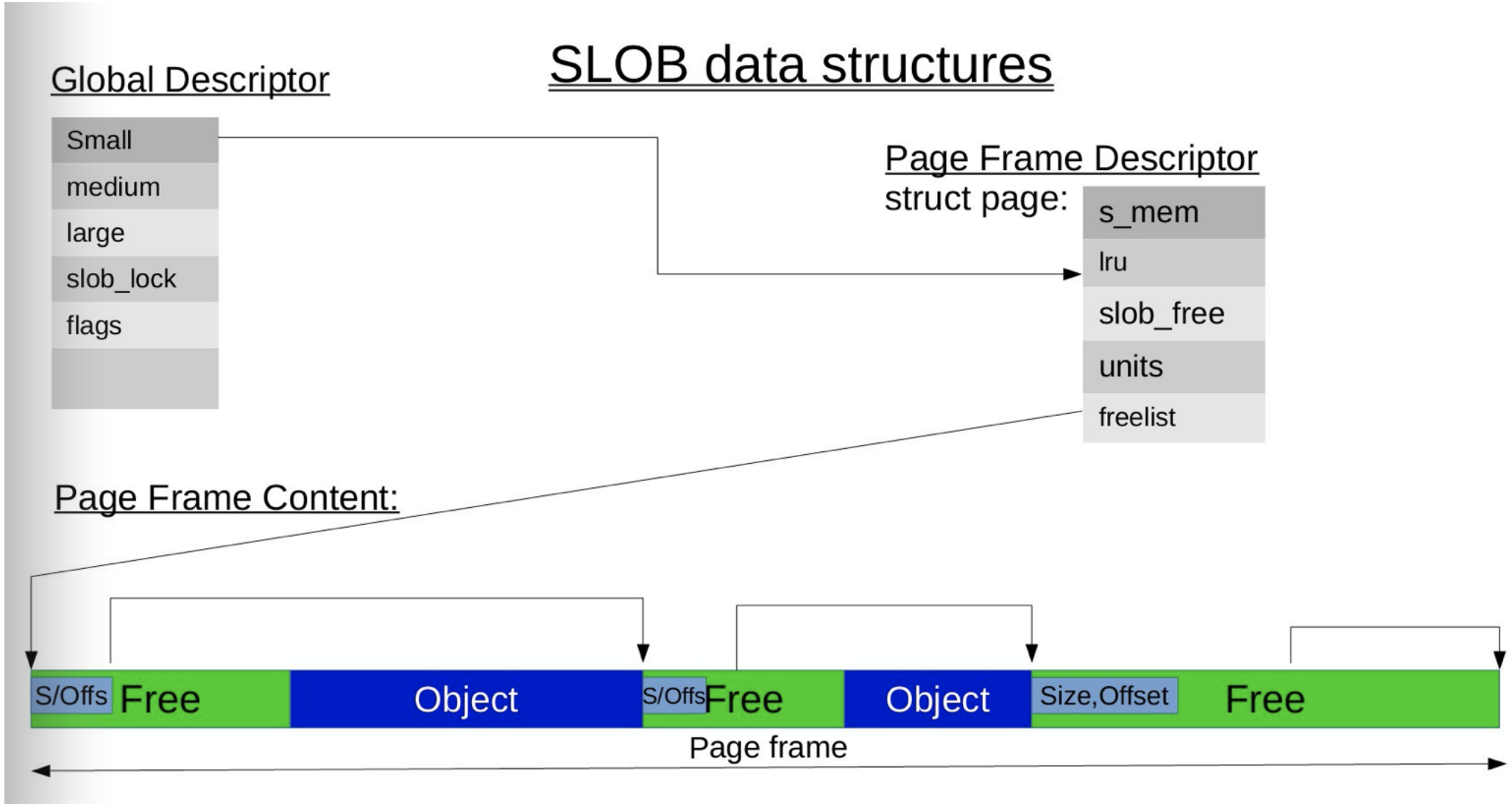
Slob

Slub의 오버헤드를 더 줄인것

페이지를 단위(SLOB_UNIT)에 맞게 짜른 뒤 블럭으로 묶어서 관리



Slob


SLOB data structures
Global Descriptor
Small
medium
large
slob_lock
flags
Page Frame Descriptor
struct page:
s_mem
lru
slob_free
units
freelist
Page Frame Content:
S/Offs
Free
Object
S/Offs
Free
Object
Size,Offset
Free
Page frame