物理内存管理

ZJUNIX

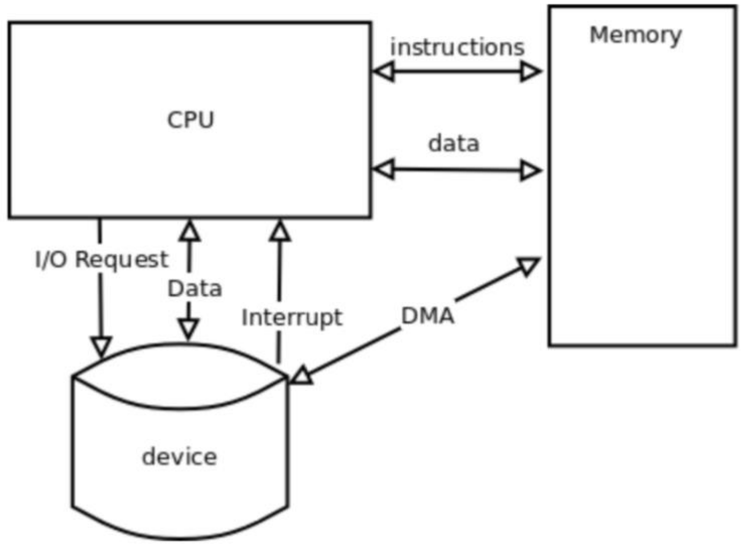
目录

物理内存管理整体设计

- 计算机系统中的内存
- 物理内存管理设计

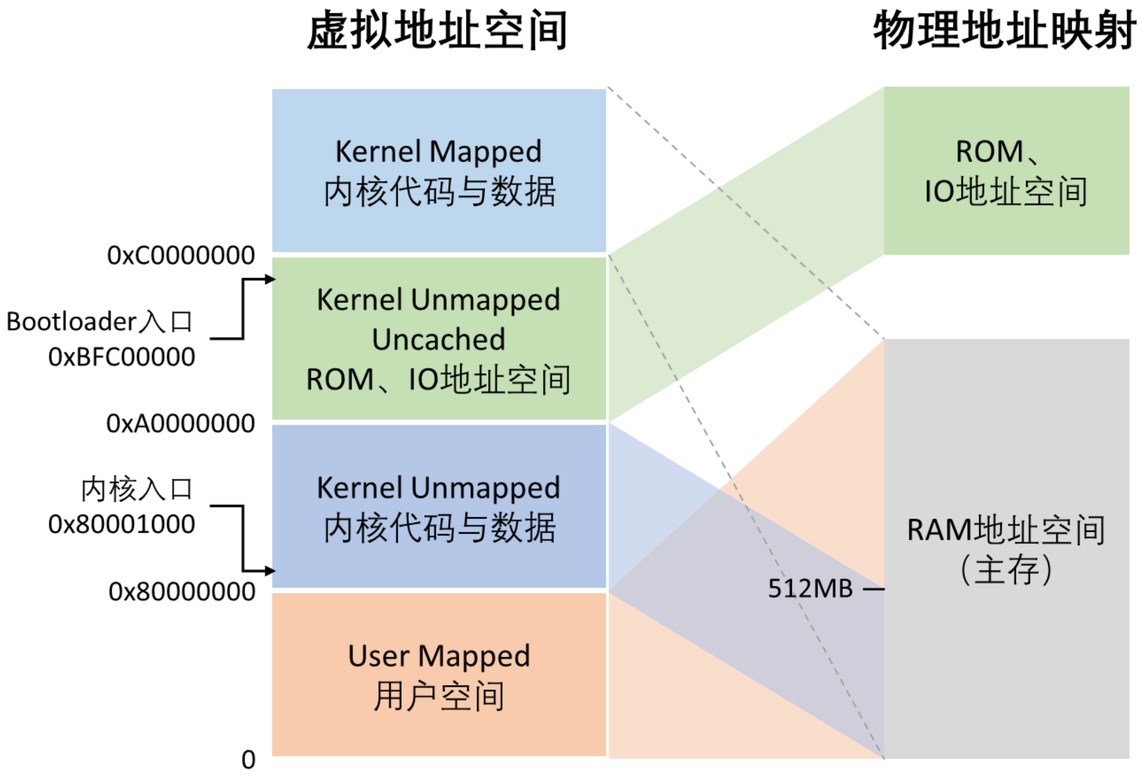
计算机系统中的内存

- 内存在存储层次结构中，介于高速缓存和外部存储之间
- 装载程序供CPU取指执行，数据运算
- 暂时存放数据，与外部存储进行数据交换



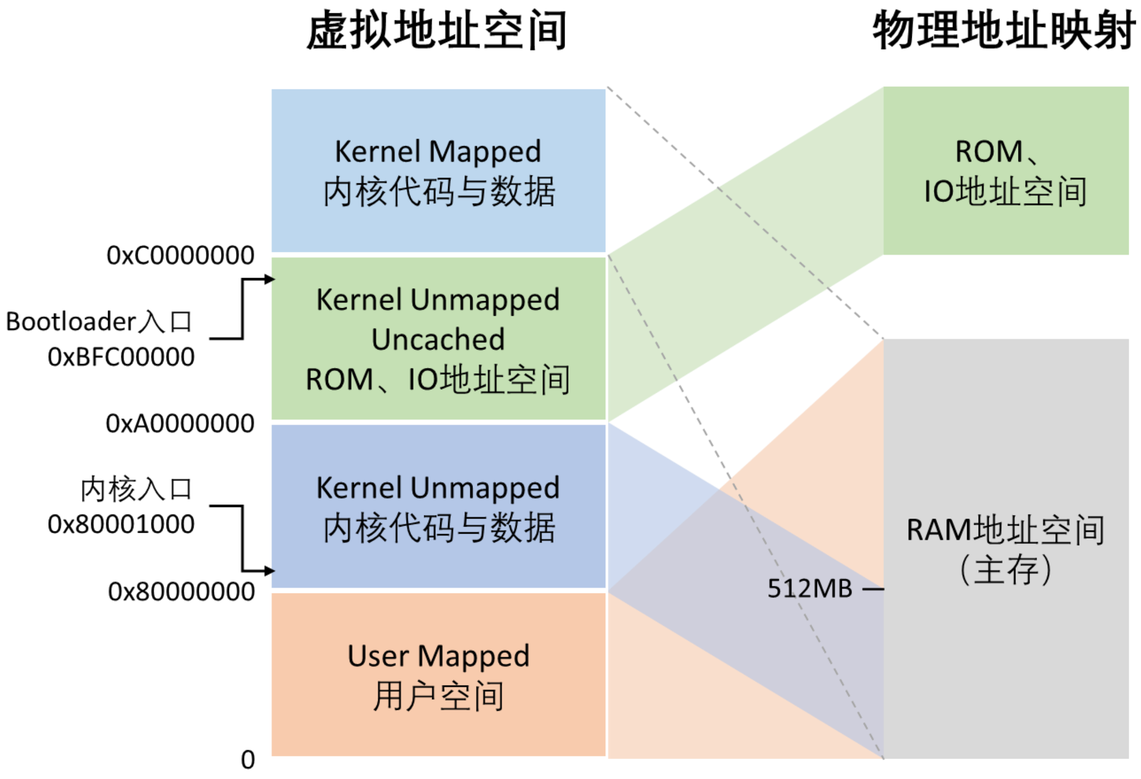
物理内存管理策略

- 根本任务：
  - 根据程序请求，动态分配、释放内存
- 基本问题：
  - 内部碎片、外部碎片



物理内存管理策略

- Bootmm：
  - 初始化物理内存信息、系统启动初期内存管理
  - 申请内存时在位图中寻找足够的连续空间
- Buddy：
  - 减少外部碎片
  - 性能较好，快速内存回收
- Slab：
  - 减少内部碎片
  - 缓存频繁分配释放的数据结构



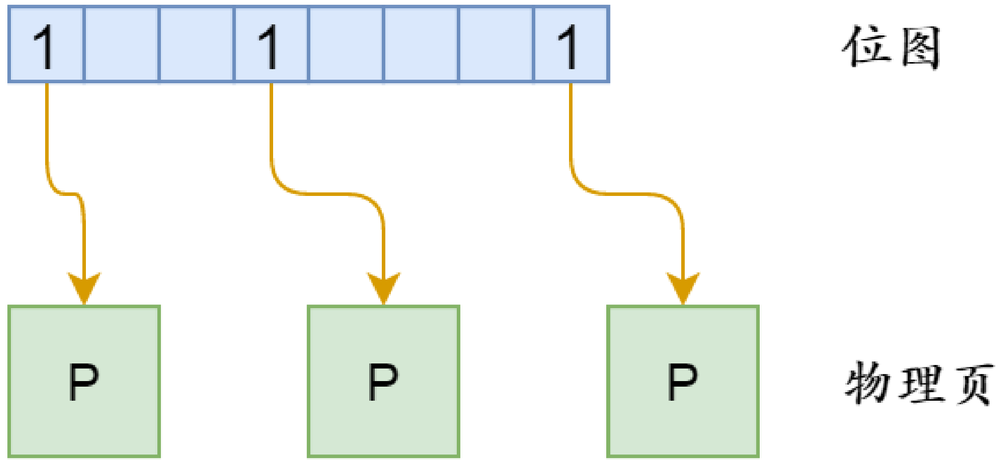
Bootmm模块设计

- Bootmm数据结构
- Bootmm初始化
- Bootmm分配与释放

Bootmm数据结构

- bootmm结构管理整个物理内存
- Phymm表示物理内存大小
- max_pfn表示系统最大物理页框号
- s_map、e_map表示内存位图的起始、末尾
- last_alloc_end表示上次分配的末尾

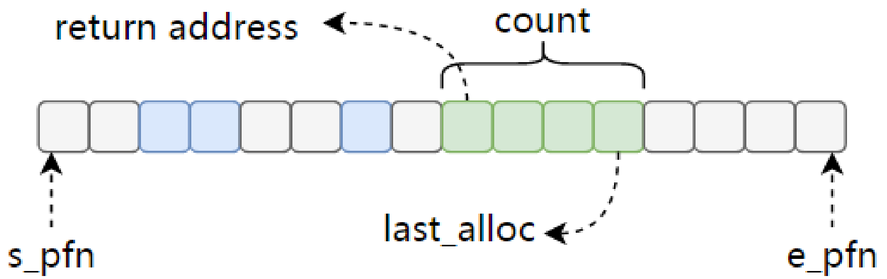
```
struct bootmm {
    unsigned int phymm;    // the actual physical memory
    unsigned int max_pfn;  // record the max page number
    unsigned char* s_map;  // map begin place
    unsigned char* e_map;
    unsigned int last_alloc_end;
    unsigned int cnt_infos;  // get number of infos stored in bootmm now
    struct bootmm_info info[MAX_INFO];
};
```



Bootmm数据结构

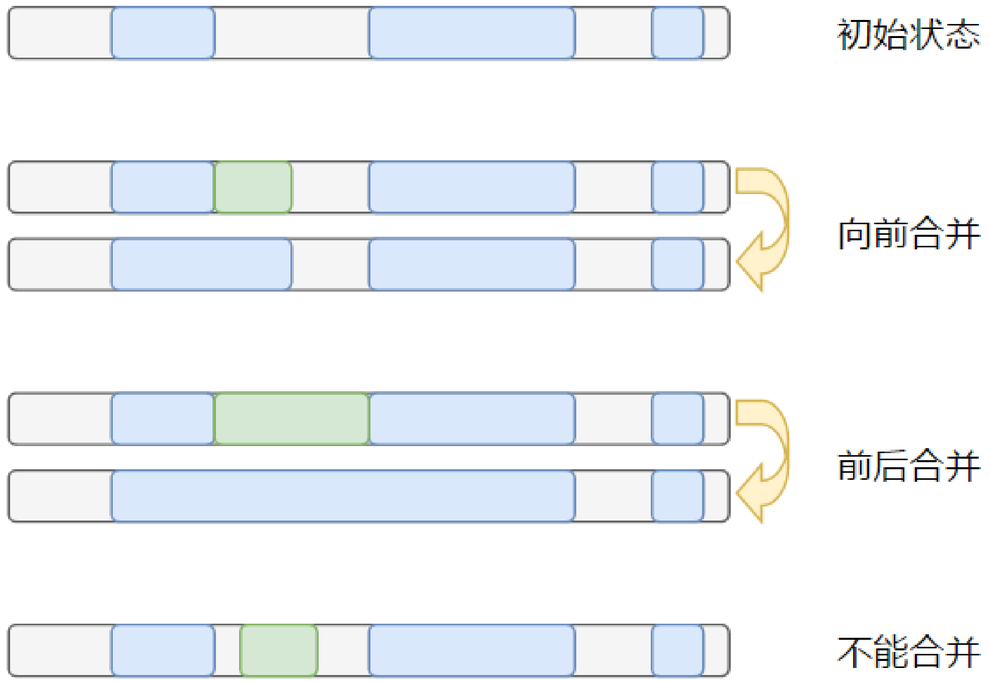
- bootmm_info结构描述一片内存区域
- start_pfn表示起始页框号
- end_pfn表示结束页框号
- type表示这段内存区域的类型

```c
// record every part of mm's information
struct bootmm_info {
    unsigned int start;
    unsigned int end;
    unsigned int type;
};
```



Bootmm初始化

- 实现插入内存区域信息
  - 遍历info数组，检查合并情况，寻找类型一致的区域
  - 如果相邻则合并信息，插入并更新结束地址，不相邻则设置新项
  - 图中蓝色为已有区域，绿色为要插入的区域
    - 向前合并，区域数不变
    - 前后合并，区域数减1
    - 无法合并，区域数加1



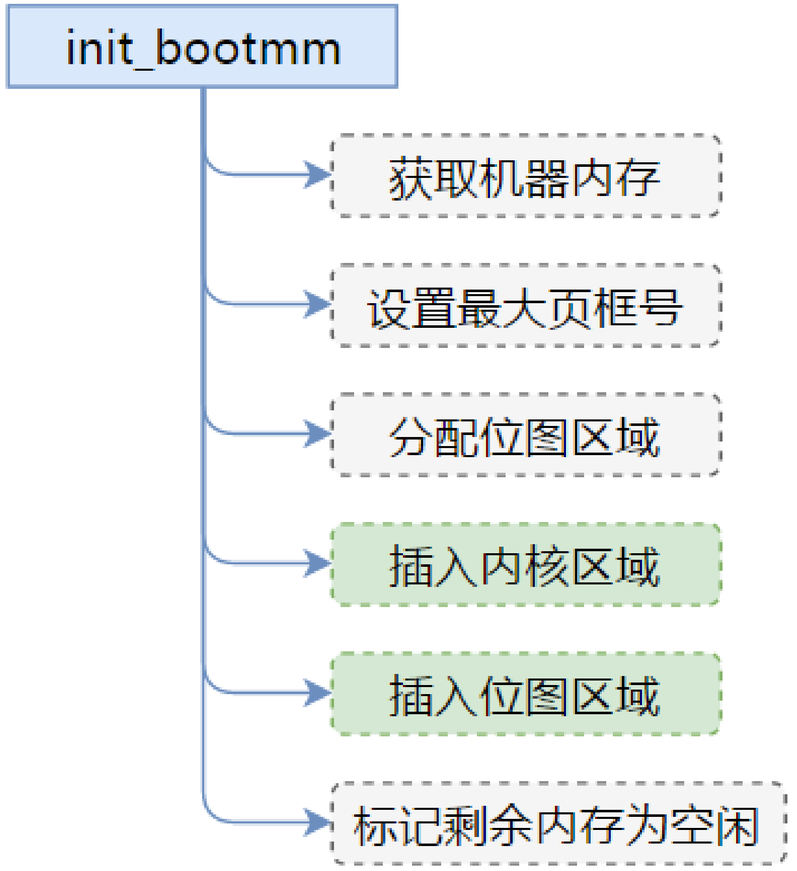
Bootmm初始化

- 实现删除内存区域信息
  - 从info数组中删除一项
  - 后续项向前搬动

```
void remove_mminfo(struct bootmm *mm, unsigned int index)
{
    unsigned int tmp;

    if (index >= mm->cnt_infos)
        return;

    for (tmp = (index + 1); tmp != mm->cnt_infos; ++tmp) {
        mm->info[tmp - 1] = mm->info[tmp];
    }

    --(mm->cnt_infos);
}
```

Bootmm初始化

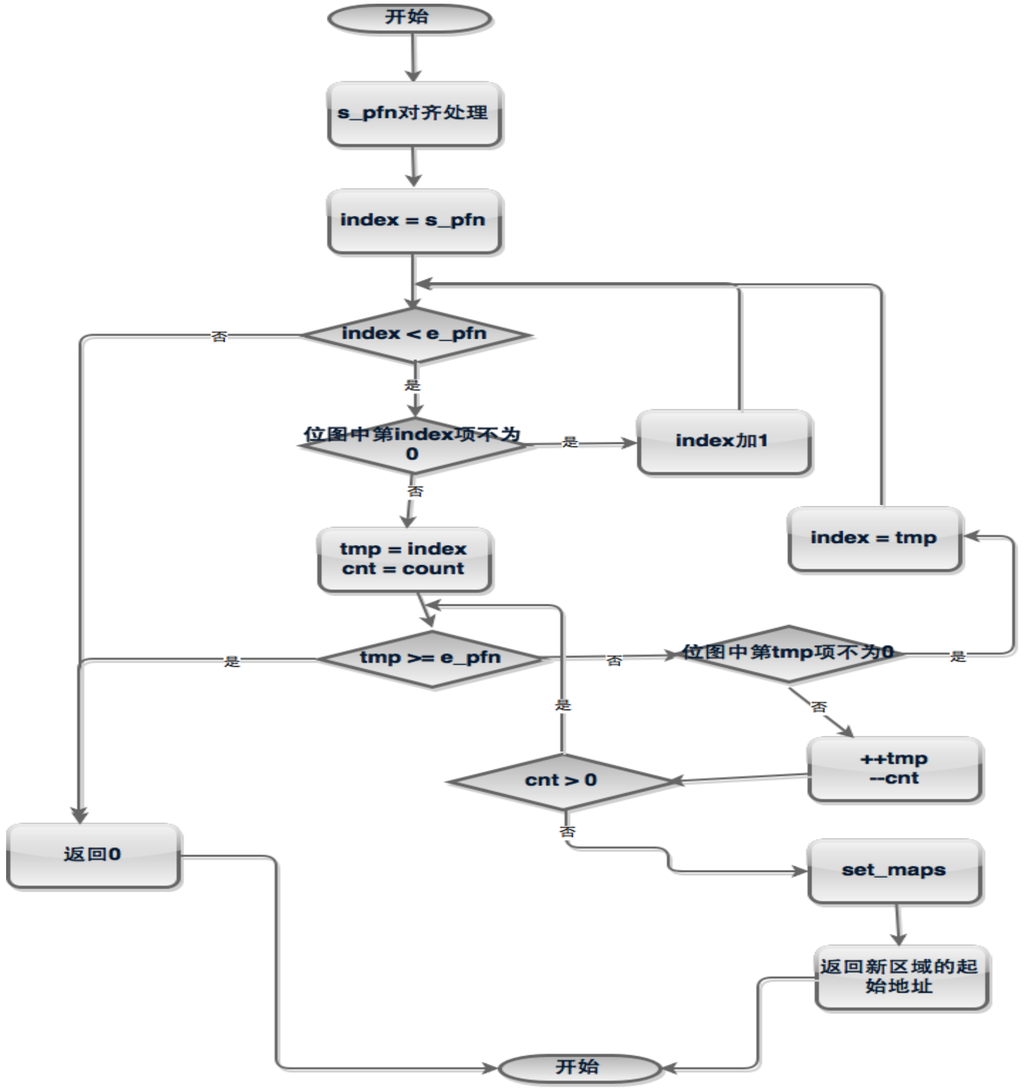
- bootmm初始化
  - 设置物理内存的相关信息
    - 物理内存大小 512MB
    - 最大页框号
  - 对内存区域使用情况进行设置
    - 内核代码/数据段：16MB
  - 初始化内存位图
    - 内核段设置为使用
    - 其余设置为空闲




```
bootmmmap));
1), _MM_KERNEL);
>> PAGE_SHIFT) - 1);

ex++) {
```

Bootmm分配与释放

- 实现find_pages
  - 根据位图情况寻找可用连续页面
  - 详细逻辑见右图
- 实现bootmm_alloc_page
  - 计算需要分配多少连续页面
  - 调用find_pages查找
  - 使用insert_mminfo插入信息



Bootmm分配与释放

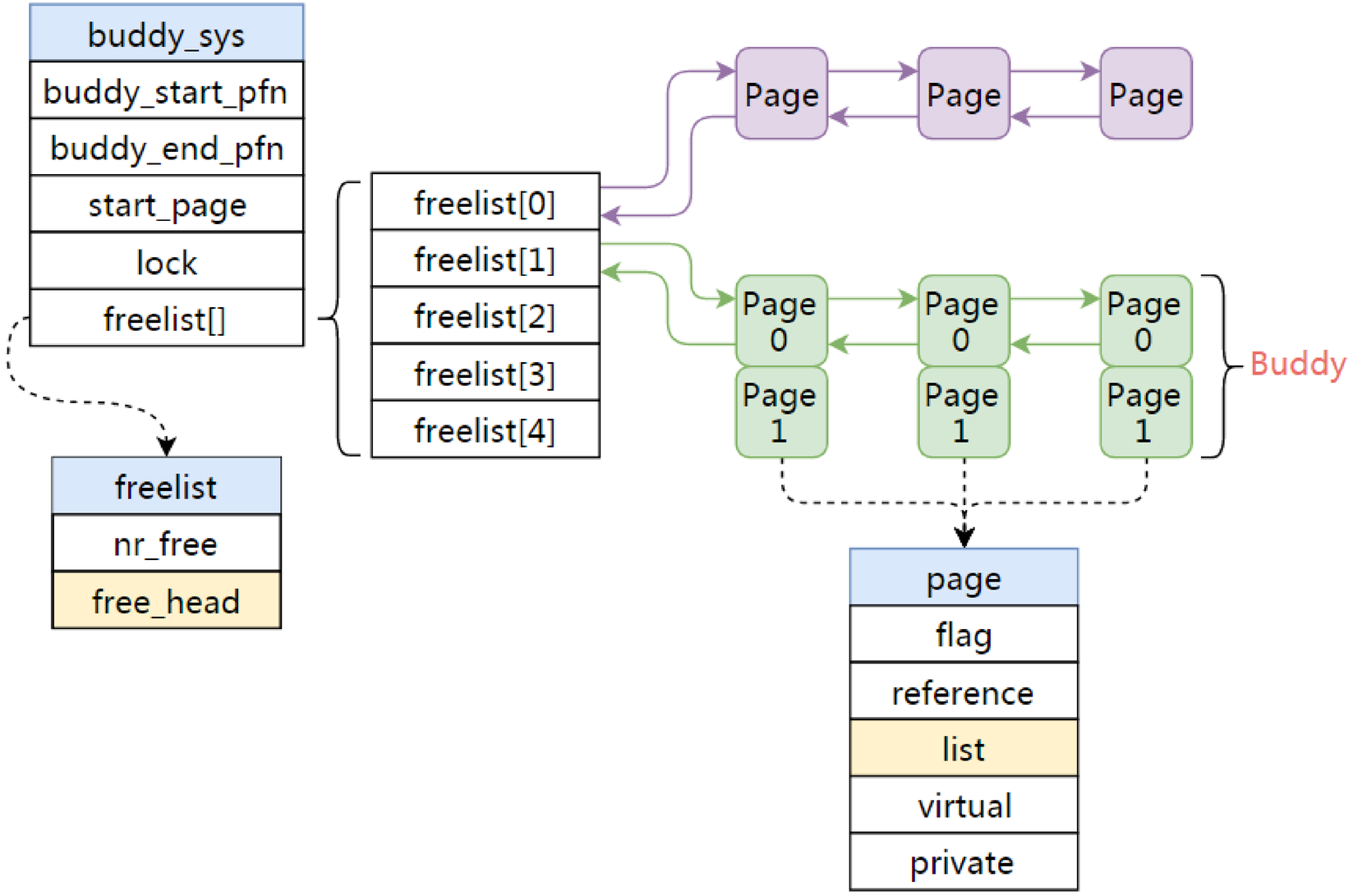
- 实现free_pages
  - 维护位图中的内存使用信息
- 实现split_mminfo
  - 维护info数组中的内存使用信息
- 事实上，内核中不应该依赖bootmm来做动态、频繁的分配释放操作
  - bootmm_alloc_page仅起到管理信息作用，每一个分配的区域都应是重要的常驻内核的数据结构。
  - 主要为buddy系统提供隔离与支持
  - 一般用途的数据结构使用静态全局变量

Buddy模块设计

- Buddy数据结构
- Buddy初始化
- Buddy内存释放
- Buddy内存分配

ZJUNIX

Buddy数据结构

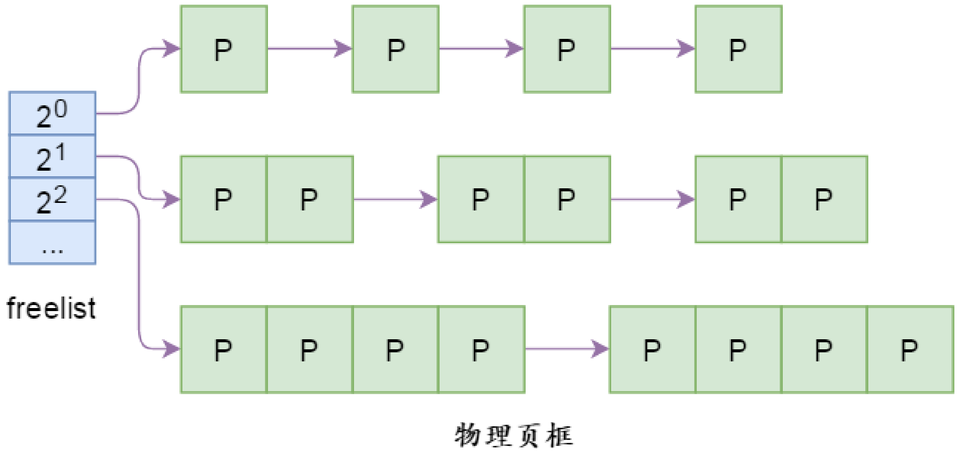


ZJUNIX

Buddy数据结构

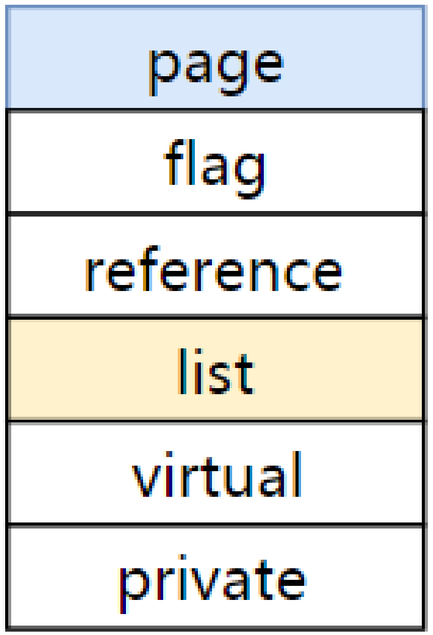
- buddy系统管理的起始页框号
- 为每个物理页面维护一个位图
- 自旋锁
- 伙伴块的空闲链表数组

```
struct buddy_sys {
    unsigned int buddy_start_pfn;
    unsigned int buddy_end_pfn;
    struct page *start_page;
    struct lock_t lock;
    struct freelist freelist[MAX_BUDDY_ORDER + 1];
};
```



Buddy数据结构

- Page
  - 描述一个物理页框

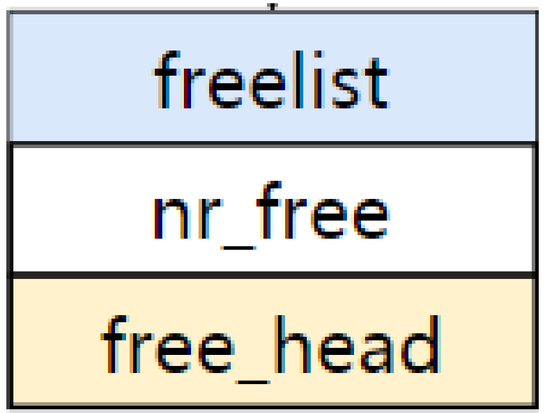



```c
struct page {
    unsigned int flag;
    int refrence;
    struct list_head list;
    void *virtual;
    int private;
};
```

ZJUNIX

Buddy初始化

- Freelist
  - 串联物理页框

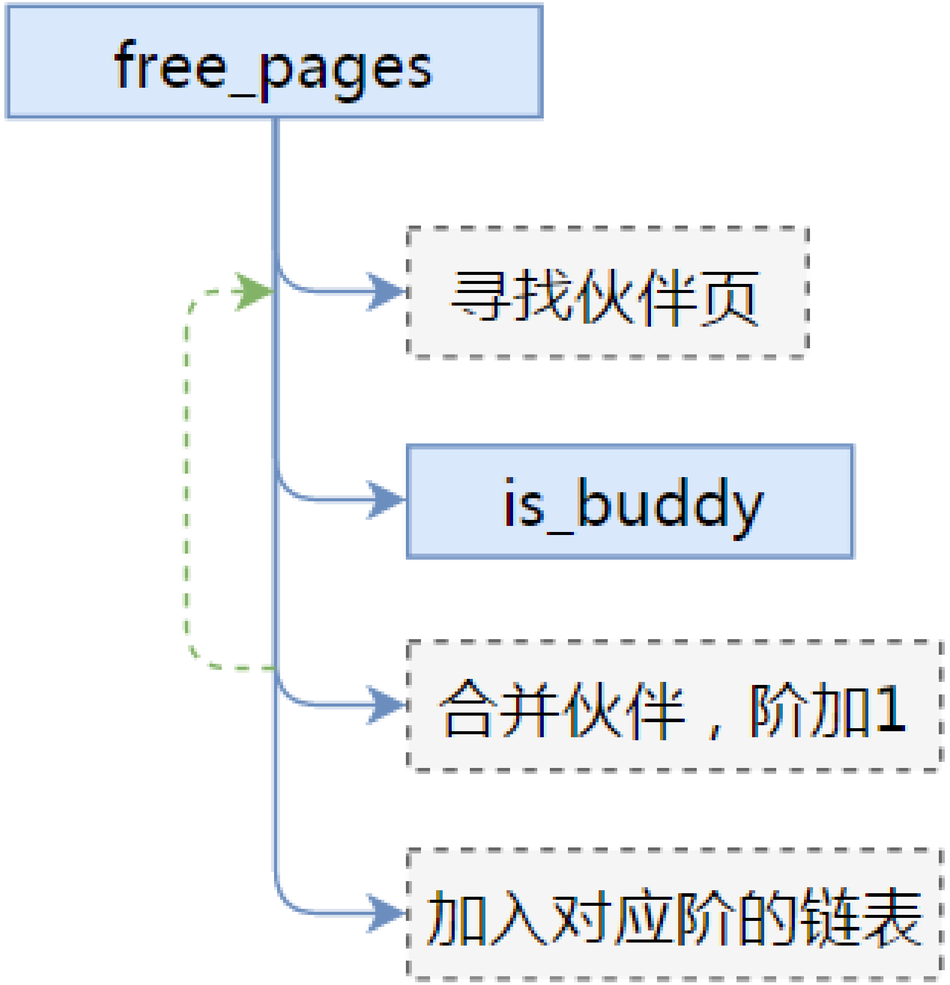



```
struct freelist {
    unsigned int nr_free;
    struct list_head free_head;
};
```

Buddy初始化

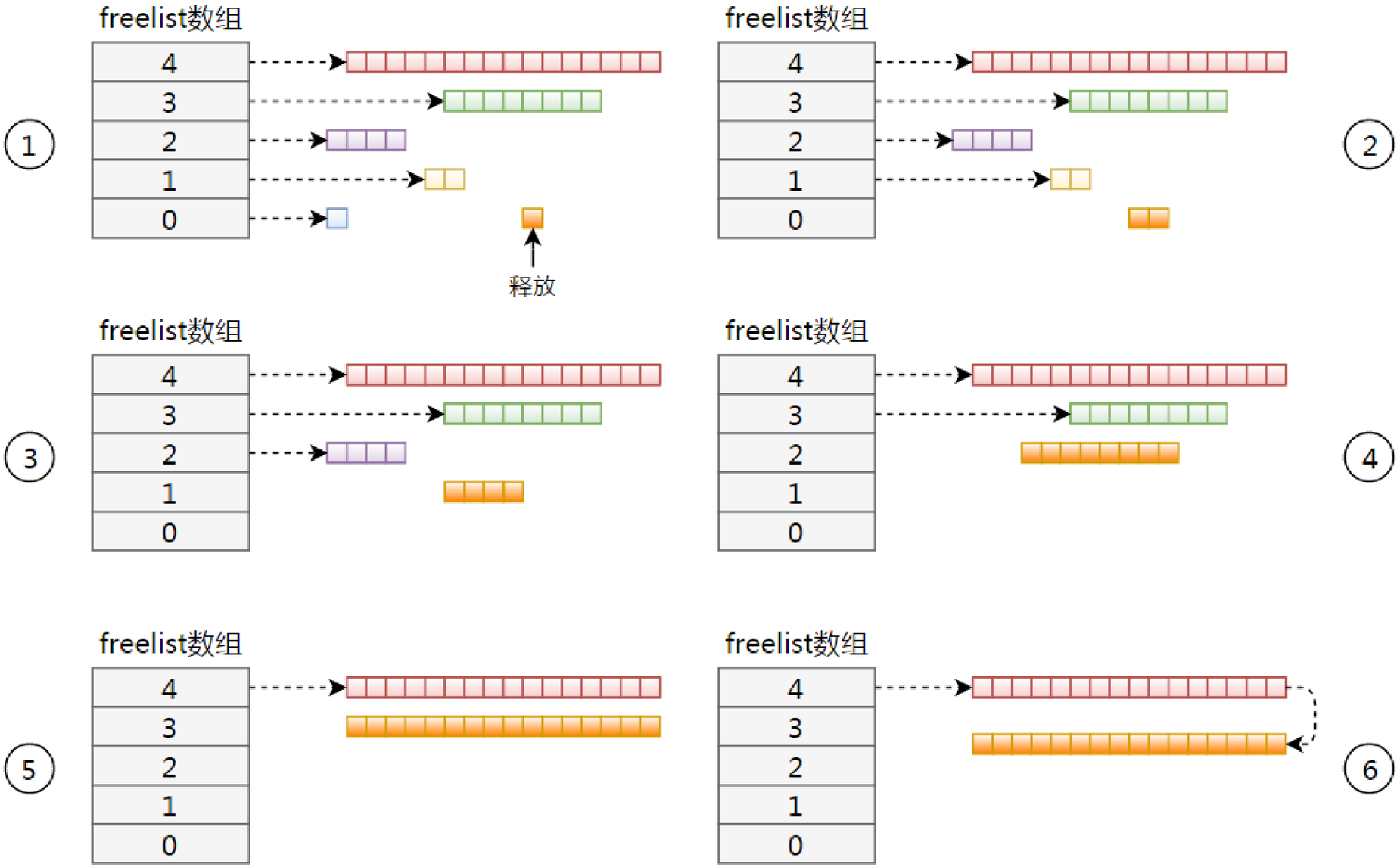
- Free_pages()将空闲页从bootmm移交至buddy系统
- 以1页的粒度释放，在释放之前buddy系统已经初始化完毕

Buddy内存释放




```c
void __free_pages(struct page *pbpage, unsigned int bplevel) {
    /* page_idx -> the current page
     * bgroup_idx -> the buddy group that current page is in
     */
    unsigned int page_idx, bgroup_idx;
    unsigned int combined_idx, tmp;
    struct page *bgroup_page;

    lockup(&buddy.lock);

    page_idx = pbpage - buddy.start_page;
    // complier do the sizeof(struct) operation, and now page_idx is the index

    while (bplevel < MAX_BUDDY_ORDER) {
        bgroup_idx = page_idx ^ (1 << bplevel);
        bgroup_page = pbpage + (bgroup_idx - page_idx);
        if (!_is_same_bplevel(bgroup_page, bplevel)) {
            break;
        }
        list_del_init(&bgroup_page->list);
        --buddy.freelist[bplevel].nr_free;
        set_bplevel(bgroup_page, -1);
        combined_idx = bgroup_idx & page_idx;
        pbpage += (combined_idx - page_idx);
        page_idx = combined_idx;
        ++bplevel;
    }
    set_bplevel(pbpage, bplevel);
    list_add(&(pbpage->list), &(buddy.freelist[bplevel].free_head));
    ++buddy.freelist[bplevel].nr_free;
    unlock(&buddy.lock);
}
```

Buddy内存释放


freelist数组
4
3
2
1
0
1
释放
freelist数组
4
3
2
1
0
2
freelist数组
4
3
2
1
0
3
freelist数组
4
3
2
1
0
4
freelist数组
4
3
2
1
0
5
freelist数组
4
3
2
1
0
6

Buddy内存分配

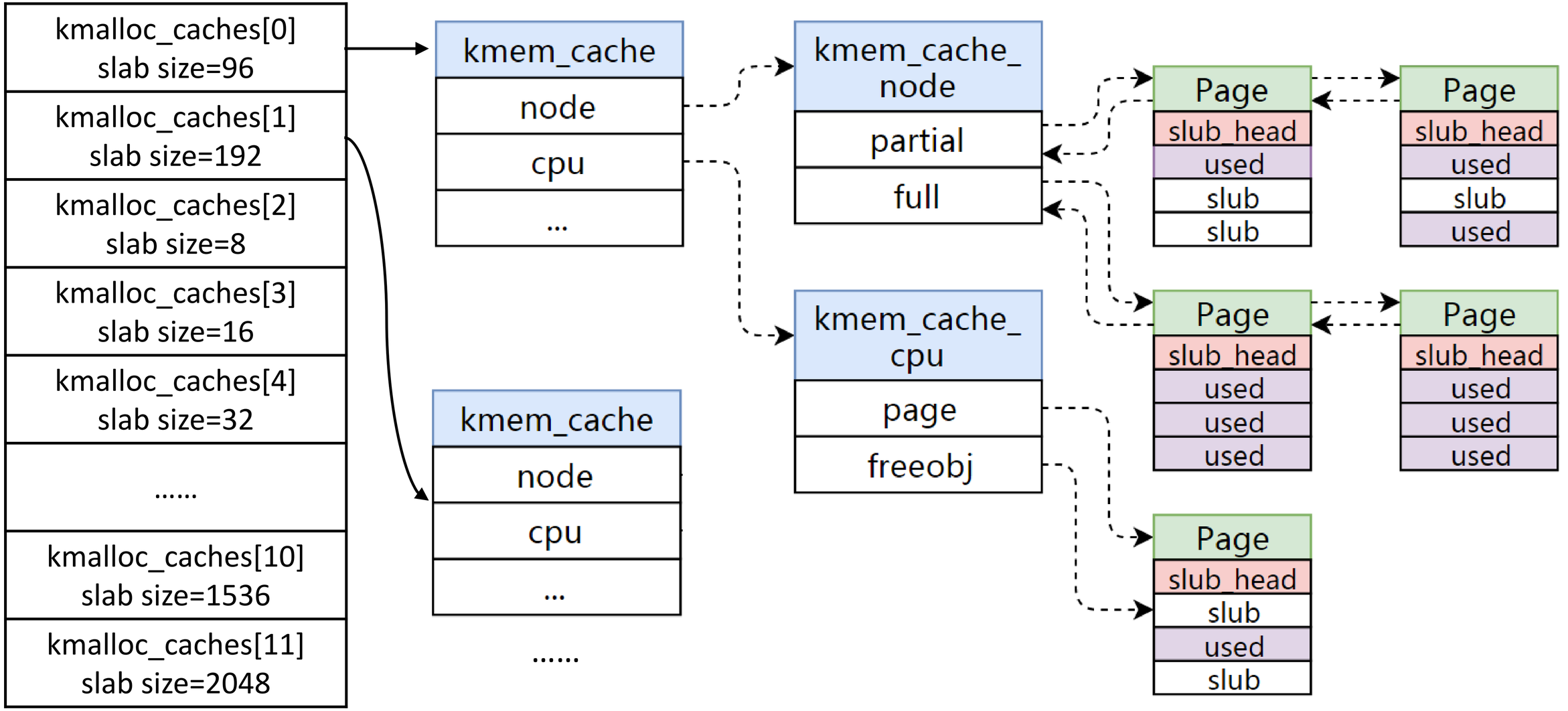
- Alloc_pages
  - 释放的逆过程
  - 从当前阶的freelist开始找，若找到，则移出freelist返回
  - 若未找到，则到更高阶寻找。高阶中找到的内存块需要分裂，除了需求的块，其它块要根据阶放到相应的free list

```
struct page *__alloc_pages(unsigned int bplevel) {
    unsigned int current_order, size;
    struct page *page, *buddy_page;
    struct freelist *free;
    lockup(&buddy.lock);
    for (current_order = bplevel; current_order <= MAX_BUDDY_ORDER; ++current_order) {
        free = buddy.freelist + current_order;
        if (!list_empty(&(free->free_head)))
            goto found;
    }
    unlock(&buddy.lock);
    return 0;
found:
    page = container_of(free->free_head.next, struct page, list);
    list_del_init(&(page->list));
    set_bplevel(page, bplevel);
    set_flag(page, _PAGE_ALLOCED);
    --(free->nr_free);
    size = 1 << current_order;
    while (current_order > bplevel) {
        --free;
        --current_order;
        size >>= 1;
        buddy_page = page + size;
        list_add(&(buddy_page->list), &(free->free_head));
        ++(free->nr_free);
        set_bplevel(buddy_page, current_order);
    }
    unlock(&buddy.lock);
    return page;
}
```

Slab模块设计

- Slab数据结构
- Slab初始化
- Slab内存分配
- Slab内存释放

Slab数据结构


kmalloc_caches[0]
slab size=96
kmalloc_caches[1]
slab size=192
kmalloc_caches[2]
slab size=8
kmalloc_caches[3]
slab size=16
kmalloc_caches[4]
slab size=32
.....
kmalloc_caches[10]
slab size=1536
kmalloc_caches[11]
slab size=2048
kmem_cache
node
cpu
...
kmem_cache
node
cpu
...
.....
kmem_cache_
node
partial
full
kmem_cache_
cpu
page
freeobj
Page
slub_head
used
slub
slub
Page
slub_head
used
slub
used
Page
slub_head
used
used
used
Page
slub_head
used
used
used
Page
slub_head
slub
used
slub

Slab数据结构

- kmem_cache_node
  - Patrial链表用来链接部分分配的slab页面
  - Full链表用来连接全部使用的slab页面
- Kmem_cache_cpu
  - 描述正用于分配的slab页面
  - Freeobj指向下一个待分配的空闲内存对象
  - page指向此页面对应的struct page

```
struct slub_head {
    void **end_ptr;
    unsigned int nr_objs;
};

struct kmem_cache_node {
    struct list_head partial;
    struct list_head full;
};

struct kmem_cache_cpu {
    void **freeobj;
    struct page *page;
};

struct kmem_cache {
    unsigned int size;
    unsigned int objsize;
    unsigned int offset;
    struct kmem_cache_node node;
    struct kmem_cache_cpu cpu;
    unsigned char name[16];
};
```

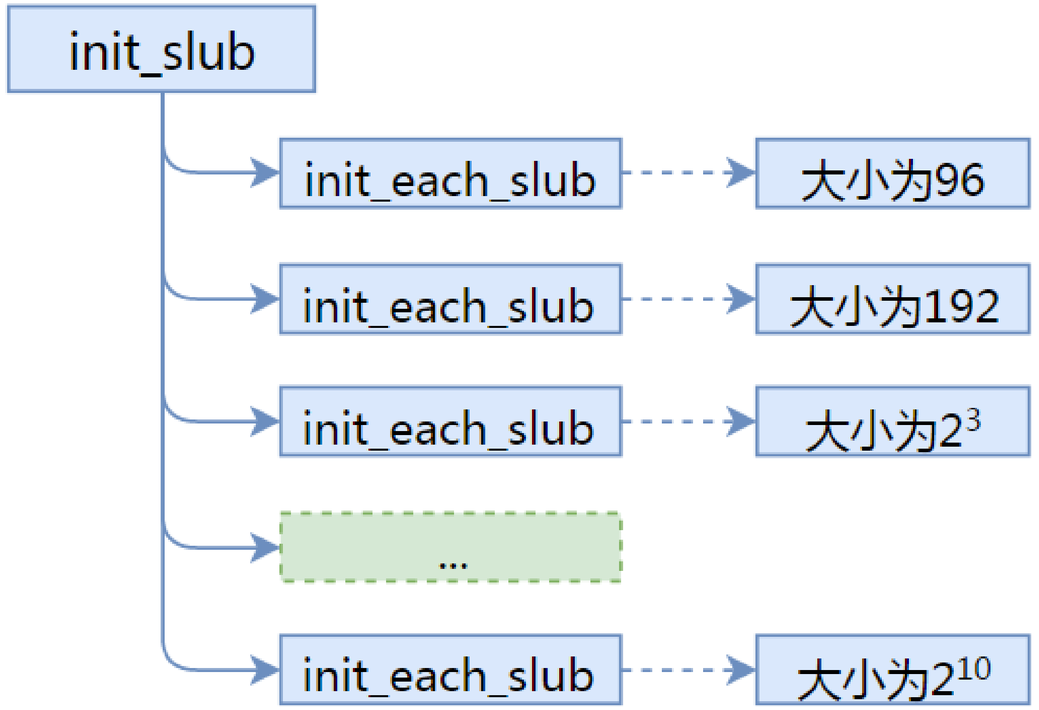
Slab数据结构

- kmem_cache
  - 维护每个slab缓存的信息
  - Size表示slab缓存中每一个对象实际占用的内存大小(包括本身和相关信息)
  - Objsize表示对象本身占用多少空间（需要对齐）
  - Offset表示该对象存放位置在slab页内的偏移
  - Node
  - Cpu

```
struct slub_head {
    void **end_ptr;
    unsigned int nr_objs;
};

struct kmem_cache_node {
    struct list_head partial;
    struct list_head full;
};

struct kmem_cache_cpu {
    void **freeobj;
    struct page *page;
};

struct kmem_cache {
    unsigned int size;
    unsigned int objsize;
    unsigned int offset;
    struct kmem_cache_node node;
    struct kmem_cache_cpu cpu;
    unsigned char name[16];
};
```

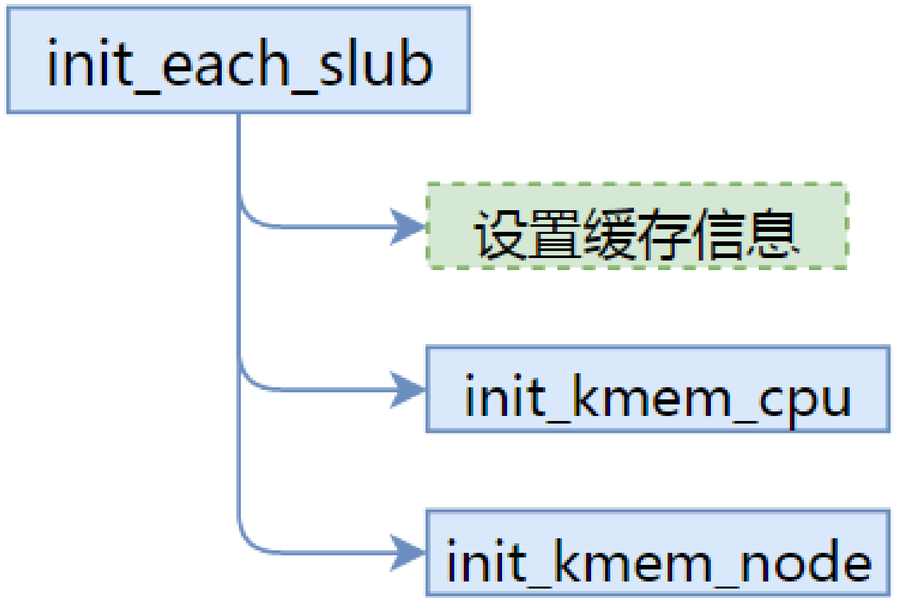
Slab初始化

- 针对一些常用大小的数据块进行分配
- 进一步，可以针对常用的内核数据结构如 struct task分配slab缓存

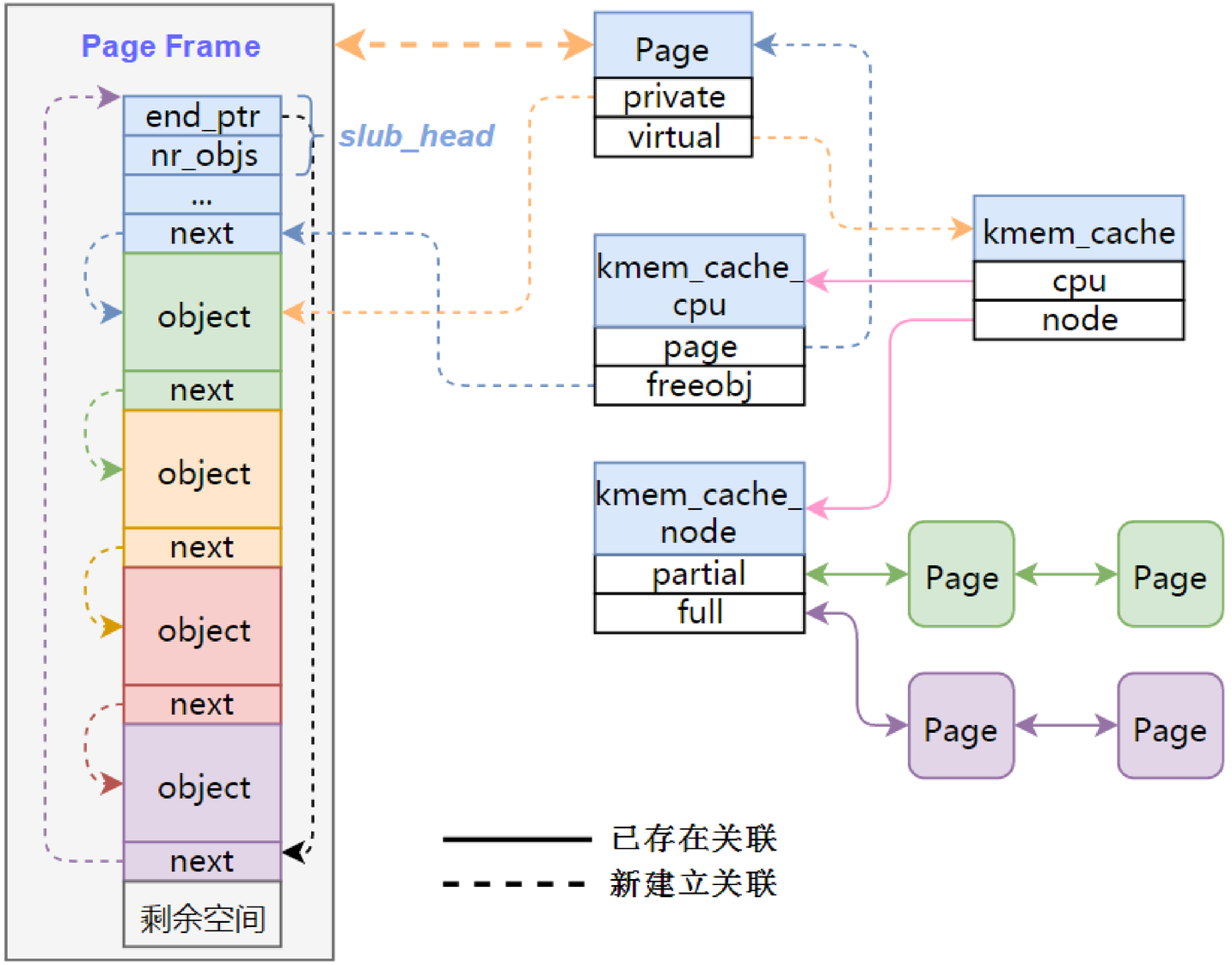


Slab初始化

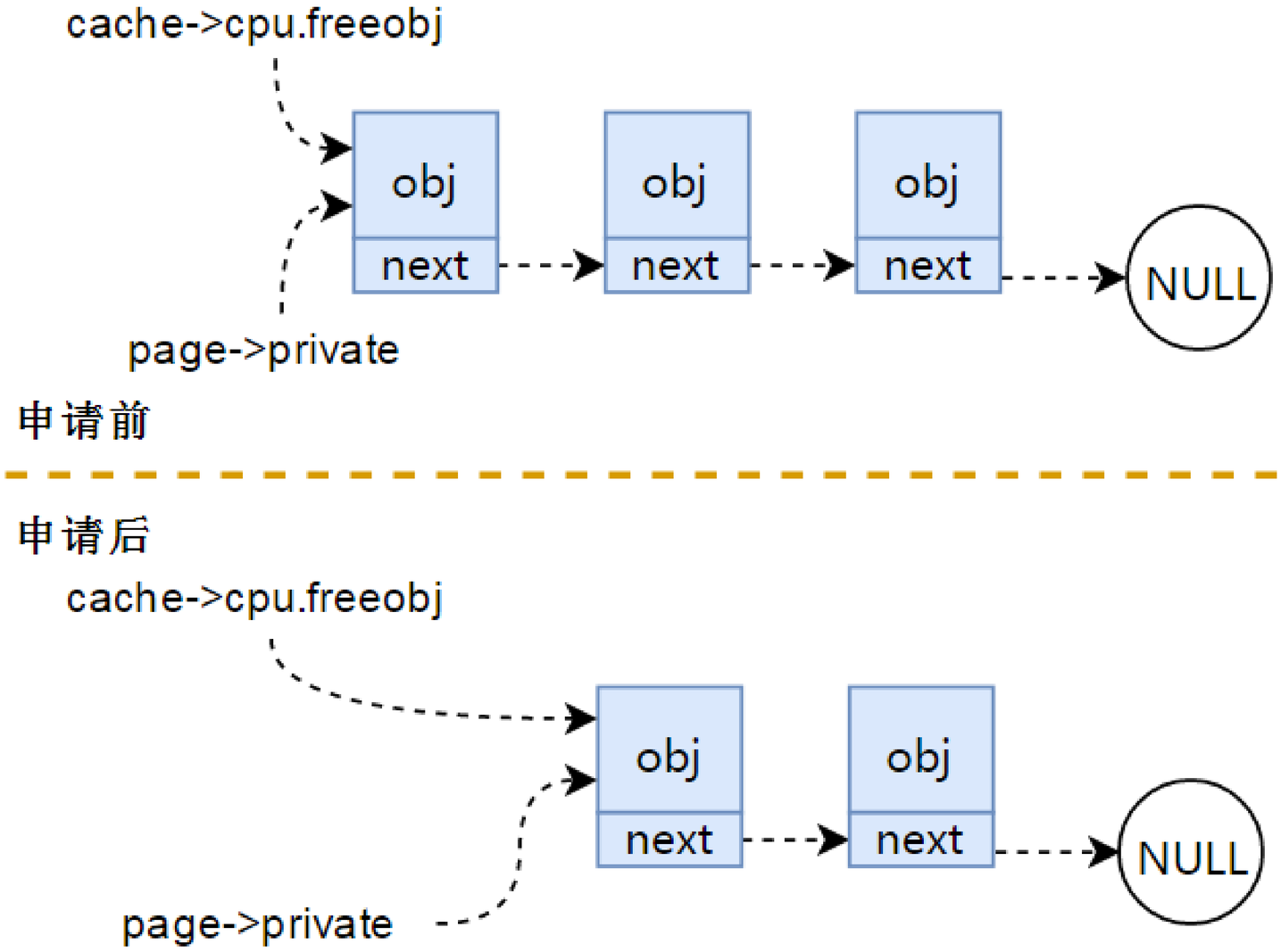
- 设置slab缓存的基本信息，如缓存对象大小
- Init_kmem_cpu
  - 初始化时不与任何页关联，第一次分配必定miss
- init_kmem_node
  - 初始化partrial链表
  - 初始化full链表



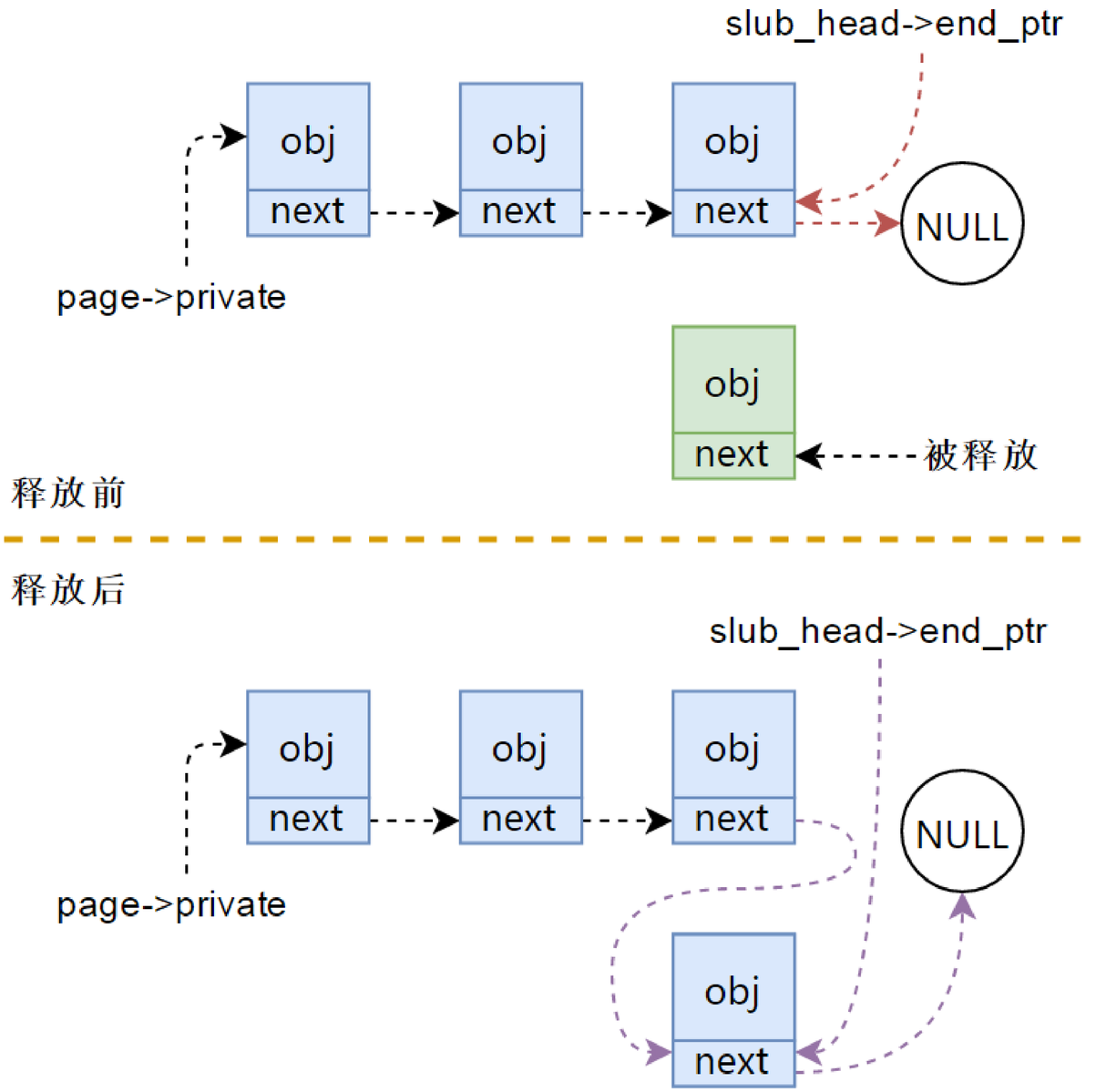
format_slub


Page Frame
end_ptr
nr_objs
...
next
object
next
object
next
object
next
object
next
剩余空间
slub_head
Page
private
virtual
kmem_cache
cpu
node
kmem_cache_
cpu
page
freeobj
kmem_cache_
node
partial
full
Page
Page
Page
Page
已存在关联
新建立关联

Slab内存分配 slab_alloc



Slab内存释放 slab_free



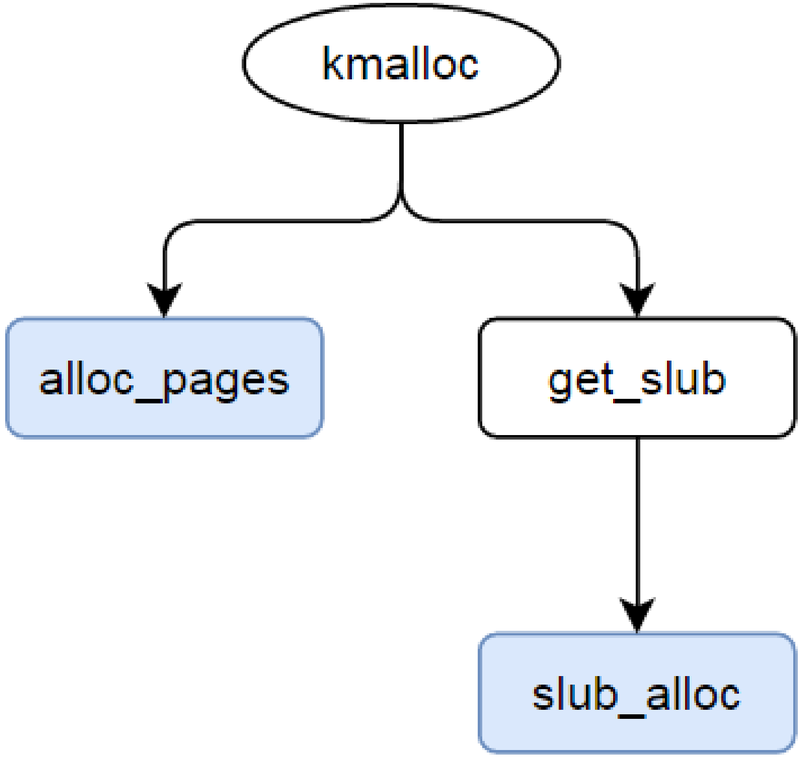
5

应用接口

- Kmalloc
- Kfree
- 使用方法

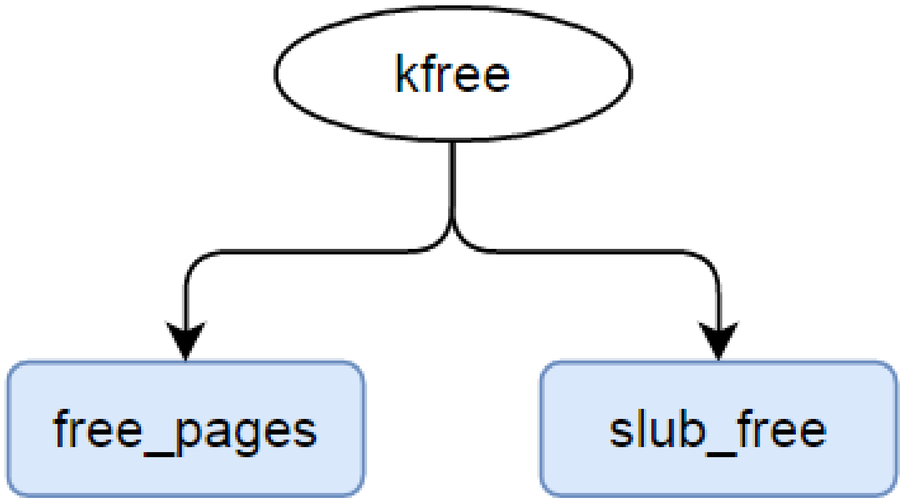
Kmalloc

- 根据申请大小，决定调用buddy或slab分配
- 大于半个页（2KB）直接调用buddy分配整数个页
- 小于2KB调用slab_alloc



kfree

- 根据_PAGE_SLAB页面标志调用buddy(free_pages)或slab(slab_free)



使用方法

- 内存管理模块仅管理物理内存
  - 虚拟内存信息由进程控制模块部分管理
- Kmalloc返回的地址为实际物理地址|0x80000000
  - 映射到kernel umapped段，方便内核逻辑内直接访问
  - 分配给内核态程序使用的内存需要根据虚拟内存信息翻译到kernel mapped 段
  - 分配给用户态程序使用的内存需要根据进程的虚拟内存信息添加相应的地址翻译，使用user mapped段
- Kfree接收的地址为内核逻辑内的地址
  - 即kmalloc返回的地址


THANK YOU