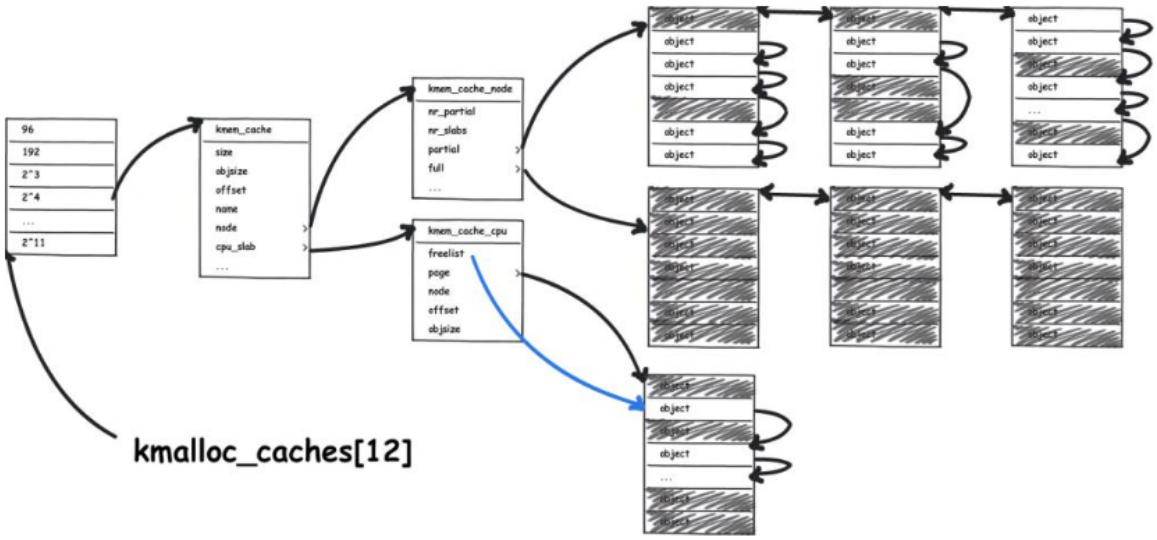
slub说明文档

Challenge 2



Slub 分配机制：
SLAB 分配器实际上是建立在伙伴系统算法之上的，SLAB 分配器使用的内存空间是通过伙伴算法进行分配的，只不过 SLAB 对这些内存空间实现了自己的算法进而对小块内存进行管理。

核心结构

kmem_cache_struct：用于描述缓存的属性。它包含了缓存的名称、对象的大小、缓存的 CPU 链表、各种指针等。

```
struct kmem_cache {
    struct array_cache *array[NR_CPUS];
    unsigned int size; // 对象的大小
    unsigned int objsize; // 对象的实际大小
    unsigned long flags;
    const char *name; // 缓存的名称
    struct kmem_cache_cpu *cpu_slab;//指向当前cpu正在使用的slab的指针
    struct kmem_cache_node *node[MAX_NUMNODES];
    struct array_cache **shared;
    int refcount;
    int size_index;
    int cpu_partial;
    int cpu_slab_nr;//slab的cpu数量
    int num;
```

```
    int gfporder;
    gfp_t allocflags;
    void (*ctor)(void *);
    int inuse;
    int align;
    int reserved;
    unsigned long min_partial;//表示最小的部分分配的slab数量
};
```

kmem_cache_cpu：每个 CPU 都有一个这样的结构，用于缓存 CPU 专用的 slab。这个结构包含了该 CPU 的一些状态信息

```
struct kmem_cache_cpu {
    void **freelist; // 空闲对象列表
    unsigned long tid; // 用于处理器识别
    struct page *page; // 当前使用的slab页
    int node;
};
```

kmem_cache_node：

```
struct kmem_cache_node {
    spinlock_t list_lock; // 用于保护全局链表的自旋锁，确保在对全局链表进行操作时，每次只能有一个线程在对链表进行增删改操作，防止多个线程同时修改全局链表导致数据不一致的情况
    struct list_head slabs_partial; // 未满的slab列表
    struct list_head slabs_full; // 满的slab列表
    struct list_head slabs_free; // 空闲的slab列表
};
```

Object：

每个 object 含有指向下一个 object 的指针

初始化：

初始化时，当前 kmem_cache_cpu 和 kmem_cache_node 中没有可用的 slab，因此只能向伙伴系统申请空闲内存页，即一个 slab，并将其分为多个 object，然后取出其中的一个 object 并标记为已被占用，接着返回给用户。

其余的 object 会标记为空闲并放在 kmem_cache_cpu 中保存，kmem_cache_cpu 中的 freelist 保存着下一个空闲 object 的地址。

简化伪代码：

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t
align, unsigned long flags,void (*ctor)(void *))
{
    // 分配用于存储kmem_cache的内存
    struct kmem_cache *cachep = allocate_memory_for_cache(); // 用于表示缓存的数
据结构

    // 初始化cachep的各个成员变量
    cachep->name = name;
    cachep->size = size;
    // 其他初始化工作

    // 申请一个空闲的内存页，作为一个slab
    struct page *new_slab = get_free_page();
    // 将这个slab分割成多个object，并加入到cachep的空闲列表中
    int num_objects_per_slab = divide_slab_into_objects(new_slab, size, align,
cachep->cpu_slab);
    // 将多余的object标记为空闲并放入kmem_cache_cpu中保存
    for (int i = 1; i < num_objects_per_slab; i++) {
        mark_object_as_free(&(new_slab->objects[i])); // 标记为空闲
        add_object_to_freelist(cachep->cpu_slab, &(new_slab->objects[i])); //
将空闲object加入到freelist中
    }

    // 将第一个object标记为已被占用并返回给用户
    mark_object_as_inuse(&(new_slab->objects[0])); // 标记为已被占用
    cachep->cpu_slab->page = new_slab; // 更新当前使用的slab
    return (void *)&(new_slab->objects[0]); // 返回给用户

    return cachep;
}
```

申请 object

1. Kmem_cache_cpu 中有空闲的 object。直接把 kmem_cache_cpu 中的下一个空闲 object 返回给用户，并把 freelist 指向下一个空闲的 object。
2. Kmem_cache_cpu 中没有空闲但 kmem_cache_node 中有空闲。此时会从 kmem_cache_node 的 partial 变量中获取有空闲的 object 的 slab 并返回给用户。并且

调一个 partial 中的 slab 换进 kmem_cache_cpu 中。

3. Kmem_cache_node 和 kmem_cache_cpu 中的 slab 已经都满了。向伙伴系统重新申请一个 slab。这个 slab 放在 kmem_cache_cpu 中，把本来的满的 slab 放到 full 链表中。

简化伪代码：

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags) {
    // 检查 kmem_cache_cpu 中是否有空闲的 object
    if (kmem_cache_cpu_has_freelist(cachep)) {
        void *obj = kmem_cache_cpu_get_freelist_obj(cachep);
        return obj;
    }

    // 检查 kmem_cache_node 中是否有部分分配的 slab
    if (kmem_cache_node_has_partial(cachep)) {
        struct page *new_slab = kmem_cache_node_get_partial_slab(cachep);
        if (!new_slab) {
            // 从伙伴系统重新申请一个 slab
            new_slab = kmem_cache_grow(cachep, flags);
            if (!new_slab) {
                // 无法申请新的 slab，返回空指针
                return NULL;
            }
        }
        kmem_cache_cpu_add_slab(cachep, new_slab);
        void *obj = kmem_cache_cpu_get_freelist_obj(cachep);
        return obj;
    }

    // 从伙伴系统重新申请一个 slab
    struct page *new_slab = kmem_cache_grow(cachep, flags);
    if (!new_slab) {
        // 无法申请新的 slab，返回空指针
        return NULL;
    }
    kmem_cache_cpu_add_slab(cachep, new_slab);
    return kmem_cache_cpu_get_freelist_obj(cachep);
}
```

释放 object

1. Kmem_cache_cpu 中缓存的 slab 就是该 object 所在的 slab。该 slab 还在 kmem_cache_cpu 中，把该 object 放到空闲列表中即可。
2. Object 所在的 slab 是 full 状态。那么释放 object 之后，该 slab 就是 partail 状态，此时把该 slab 添加到 kmem_cache_node 中的 partial 链表中。
3. Slab 是 partial 状态。直接把该 object 加入到该 slab 的空闲队列中。
4. Object 在释放后，slab 中的 object 全部是空闲的。此时还需要把整个 slab 释放掉。

简化伪代码：

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp) {
    struct kmem_cache_cpu *c;
    struct page *page;
    struct kmem_cache_node *n;

    c = this_cpu_ptr(cachep->cpu_slab);

    // 释放 object
    // 如果该 slab 在 kmem_cache_cpu 中，将 object 放到空闲列表中
    if (c->page && PageSlab(c->page) && objp >= c->page->s_mem && objp < (void
*)c->page->s_mem + cachep->size) {
        // 将 object 放入空闲列表
        put_cpu_partial(cachep, objp);
    }
    // 如果该 slab 是 full 状态，将该 slab 添加到 kmem_cache_node 的 partial 链表
中
    else if (PageSlab(c->page) && c->page->objects == cachep->num) {
        // 添加该 slab 到 kmem_cache_node 的 partial 链表中
        n = get_node(cachep, c->page);
        if (list_empty(&c->page->lru))
            list_add(&c->page->lru, &n->slabs_partial);
    }
    // 如果该 slab 是 partial 状态，将 object 加入到该 slab 的空闲队列中
    else if (PageSlab(c->page)) {
        // 将 object 加入到该 slab 的空闲队列中
        slab_free(cachep, objp, c->page);
    }
    // 如果该 slab 中的 object 全部是空闲的，释放整个 slab
    else if ((page = virt_to_page(objp)) && PageSlab(page) && page->objects ==
0) {
        // 释放整个 slab
        slab_destroy(cachep, page);
```

```
    }
}
```