操作系统Lab2

首先对内存布局进行总体上的介绍：下面是pmm初始化后的内存布局，其中可用物理空间是从0x80000000到0x88000000(Challenge3中可以观察到)，但是涉及到虚拟地址映射的位置是从kernel开始的，kernel起始位置对应的虚拟地址是0xffffffffc0200000(对应物理地址是0x80200000)。具体地址都是通过在pmm.c文件中附加cprintf输出并进行观察得出。

注意：下面没有包含opensbi的空间(0x80000000到0x80200000)，是因为这段是在M模式下完成的，还没有涉及到虚拟地址。而且这个布局仅适用于基础部分，在后续的buddy和slub中内存布局会发生变化。

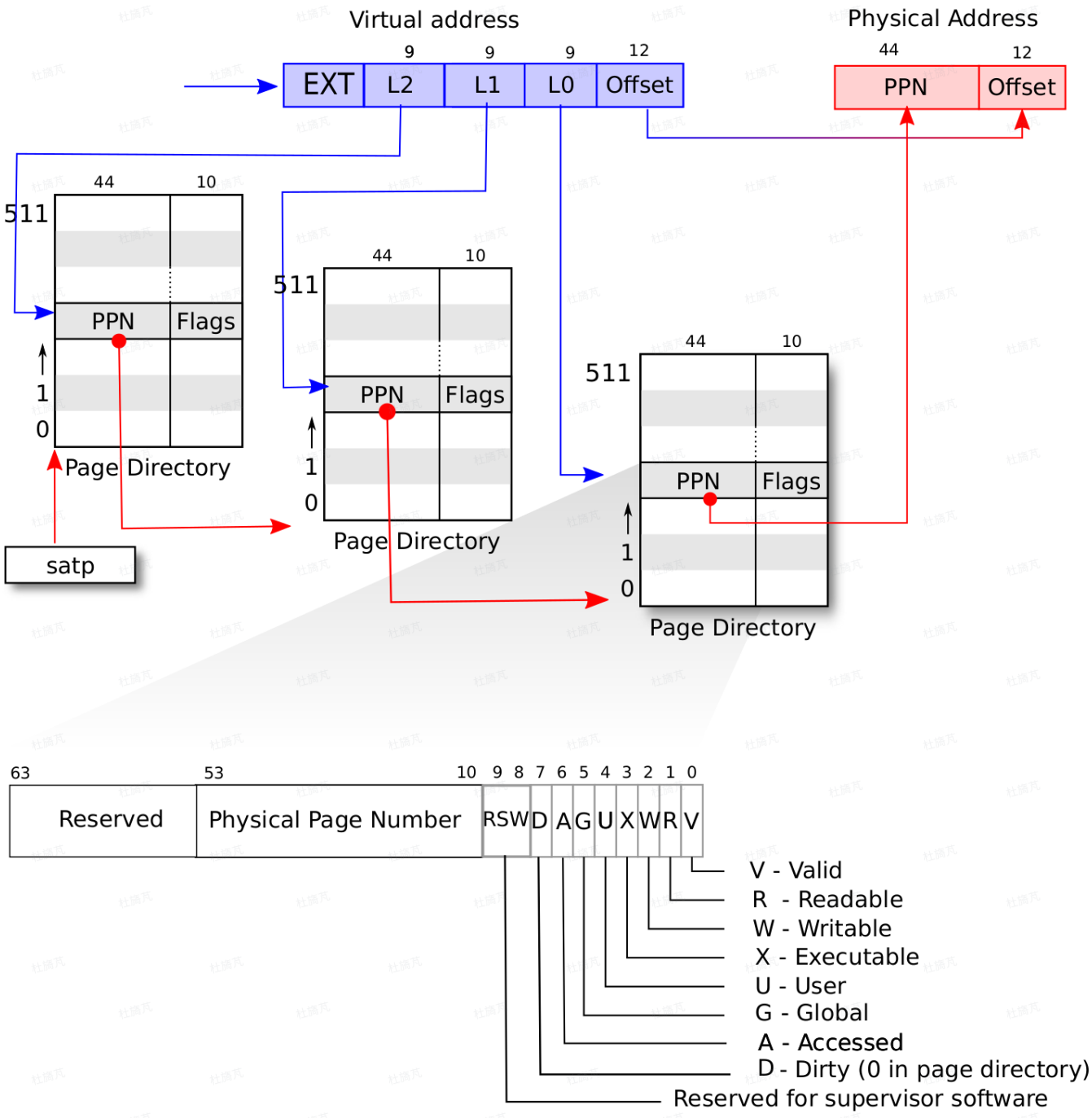
```
+------------------+ <-- vma: 0xffffffffc0200000
|                  |     pma:         0x80200000
|      KERNEL      |
|                  |
+------------------+ <-- pma:         0x80206470
|       HOLE       | ----------------> 为了对齐
+------------------+ <-- pma:         0x80207000
|                  |
|       Pages      | ----------------> 存储页结构
|                  |
+------------------+ <-- mem_begin (要向上对齐，但是无影响)
|                  |     pma:         0x80347000
|       Mem        |
|                  |
+------------------+ <-- mem_end (要向下对齐，但是无影响)
                        pma:         0x88000000
```

补充：这里在利用gdb工具进行调试的时候，发现在经过刷新TLB指令后pc就无法正确的获取到指令地址，会不断报错，于是针对这个问题，我们进行了探索，发现由于，该指令执行后pc没有进行更新，于是发生了缺页异常，然后观察stvec寄存器，其值为0x80200000，也就是内核起始地址，于是跳转到0x80200000继续执行，但是发现由于原来页表项只映射了高地址，而没有映射低地址，于是这里我们通过直接使用物理地址的值（映射到了自身）来进行处理，由于0x80000000对应的顶级页表的页号是00000010，也就是顶级页表的第三个页表项，于是对entry.S文件的下列代码进行更改，下面展示更改后的代码：

```
# 分配 4KiB 内存给预设的三级页表
boot_page_table_sv39:
    # 0xffffffff_c0000000 map to 0x80000000 (1G)
```

```
    .zero 8 * 2
    .quad (0x80000 << 10) | 0xcf
    .zero 8 * 508
    # 设置最后一个页表项，PPN=0x80000，标志位 VRWXAD 均为 1
    .quad (0x80000 << 10) | 0xcf # VRWXAD
```

练习1：理解first-fit 连续物理内存分配算法



上图是sv39的一个虚拟地址映射过程。

设计实现过程：

分析代码总体结构可以看出，程序在进行物理内存分配的过程中，首先在**kern_init()**中调用了**pmm_init()**来**初始化物理内存管理**和**对虚拟内存映射**。在其中先后调用了**init_pmm_manager()**和

page_init()两个函数，前者负责简单地确定**物理内存管理器pmm_manager** 和调用对应管理器中的默认初始函数（以first-fit为例，实现**管理页面的双向链表**的初始化和总内存块数）；后者负责探测物理内存状态，将空闲内存以规定的页面大小创造页面结构，完成 **空闲内存起始地址** 和 **可使用页面数** 的计算后，调用封装好的**init_memmap()**来进一步初始化管理器。

作为段页式内存管理基础，有必要先介绍页结构体Page:

```
struct Page {
    int ref;                        // page frame's reference counter
    uint64_t flags;                 // array of flags that describe the status c
    unsigned int property;          // the num of free block, used in first fit
    list_entry_t page_link;         // free list link
};
```

- ref为该页帧的引用计数器，也就是映射此物理页的虚拟页个数，在增加映射时递增，取消映射时递减；
- flags是描述该页状态的标记数组，其含有PG_reserved和PG_property两个成员，前者表示该页是否被内核占用（1/0），后者表示该页是否为一段空闲内存块的首页(1/0)。
- property比较特殊，其记录连续空闲页的数量。该变量仅在PG_property标志位为1时有效，作为空白页面分配时的重要参数。
- page_link是把多个连续内存空闲块链接在一起的双向链表指针。连续内存空闲块利用这个页的成员变量page_link来链接比它地址小和大的其他连续内存空闲块，在分配，释放过程都起作用。

然后我们来介绍**first-fit算法**中的方法：

首先，first-fit算法宏定义了两个关键参数，分别为双向链表结构 free_list 和空闲页面数量 nr_free 。

first-fit算法的具体实现主要体现在具体初始化函数default_init_memmap()，分配页面函数default_alloc_pages()和 释放页面函数default_free_pages()中，下面对其分别进行分析：

- void default_init_memmap(struct Page *base, size_t n)

传入空闲页面起始地址base和空闲页面数量n，初始化每个物理页面属性，然后将全部的可分配物理页视为一大块空闲块加入空闲表。

- struct Page * default_alloc_pages(size_t n)

首次适配算法要求按照地址从小到大查找空间，所以空闲链表中的页面需要按照地址从小到大排序。这样，首次适配算法查询特定空闲空间的方法就是从链表头部开始找到第一个符合要求的空间，并将这个空间从空闲表中删除。空闲空间在分配完要求数量的物理页之后可能会有剩余，那么需要将剩余的部分作为新的空闲空间插入到原空间位置（这样才能保证空闲表中空闲空间地址递增）。

- void default_free_pages(struct Page *base, size_t n)

先通过遍历将页面属性调整为空闲状态，更改头页面（也就是base）的property值和总共空闲页的数量，再利用双向链表的前后驱函数探测前后页面的空闲状态。如果有空闲则进行页面合并，将尾页并

入头页，修改对应页的property值，并从链表中摘除尾页。

你的first-fit算法是否有进一步的改进空间?

可以有以下的优化：

1. **优化内存的利用率：**first fit算法会产生大量小的碎片内存块,浪费空间。可以设定一个固定周期合并可以合并的内存碎片，例如进行100次内存分配就遍历进行合并一次，将其合并到相邻的空闲内存块中，减少碎片。

这个优化可以有效减少内存碎片。

测试

测试部分是由源码中已经给好的测试函数，这里不进行赘述。

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

设计实现过程：

调用过程与first-fit完全相同，唯一的区别就是在分配空间时的算法不一致，best-fit方法需要找到满足条件但却最小的那个空间，于是需要遍历整个页链表，并维护一个**min_size**保存选中的最好的页面。

这部分需要进行编程，但是除了一个地方有区别，剩下的部分都与first-fit算法实现一致，有区别的代码块如下：

```
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
     /*LAB2 EXERCISE 2: YOUR CODE*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 遍历空闲链表，查找满足需求的空闲页框
    // 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            page = p;
            min_size = p->property;
        }
    }
```

代码中用min_size去存储了当前找到的最小连续空闲页框数量，注意，这里原来在first-fit方法中存在的break要去掉，因为需要去遍历全部free_list才能确定哪个是最小的，而不是找到一个就跳出。

你的 Best-Fit 算法是否有进一步的改进空间?

可以有以下的优化：

1. **碎片管理**：同样的，best-fit也有可能导致内存碎片化，尤其是当存在大量小块内存请求时。同样可以通过类似上文first-fit改进思路的方法进行改进。
2. **缓存和索引**：在纯粹的best-fit策略中，每次都要遍历整个空闲块列表以找到最佳匹配。这可能非常低效。可以考虑引入索引或缓存机制，例如使用平衡树或跳表来快速查找适合的空闲块。可以优化原来时间复杂度为O(n)的遍历搜索算法。
3. **延迟合并**：一旦释放了内存块，就立即查找相邻的空闲块并合并它们。但这可能不是每次释放都需要的。可以考虑在空闲块数量达到一定阈值或在某些特定时机再执行合并，这样可以减少频繁的合并操作。
4. **预测和预分配**：基于程序的内存使用模式，可以尝试预测未来的内存需求，并预先分配一块更大的空闲块，以减少后续的搜索时间。

测试

测试部分是由源码中已经给好的测试函数，这里不进行赘述。

Challenge 1：buddy system 分配算法

首先根据伙伴分配器的极简实现理解其实现原理：

1. 把物理内存抽象成一个完全二叉树,每个节点代表一块内存。
2. 节点结构体有 size 和 longest 两个变量，分别代表原有内存和可分配的最大内存。
3. 叶节点 size 相同,是系统分配的最小内存块（本实验实现中，**叶节点**大小为一页）。非叶节点 size 为左右子节点的两倍。
4. 内存分配时,从根开始按需求使用 **buddy2_alloc 算法**找到可分配节点,更改其和其父节点的longest值。
5. 内存释放时，释放本节点后检测伙伴节点间是否可以合并，如果可以则递归向上合并，调整longest值。

伙伴系统的实现通过维护运算简单的二叉树高效实现分配和释放，通过递归分割和合并内存块提高内存利用率，减少外部碎片。而该极简设计的实现通过longest数组的使用简化了判断节点是否空闲、伙伴节点是否可以合并等状态，**既可以表示权重又可以表示状态，**非常高效。

针对本实验物理内存管理的原有结构，我们在 buddy system 的基础上进行以下适配：

- 在原有管理页面的双向链表 **free_list** 基础上加入 buddy2 结构体，实现在链表上层的完全二叉树建构；
- 建立 buddy2 数组， 将 longest 由可扩展的数组改为变量，防止扩展带来的语法影响；
- 新建 allocRecord 结构体数组来记录分配块的信息，主要实现页面使用标记对双向链表中页的匹配；相对应的记录 已分配块数的数量。

下面分别介绍初始化，分配和回收三个关键函数的实现：

- static void buddy_init_memmap(struct Page *base, size_t n)

首先按照提供的初始页面地址和页面数量对双向链表进行初始化，与之前 first/best fit 不同的是单页放置（p->property = 1）与伙伴系统配合；接着调用buddy2_new()函数对二叉树节点初始化，从根节点开始，根据输入的页面数（已调整为2的幂次）进行 size 和 longest 的赋值。

- static struct Page* buddy_alloc_pages(size_t n)

通过调用buddy2_alloc()函数获取目标页首的偏移量，通过对底层双向链表的的遍历找到以偏移量为起始的page**并用rec项记录**（以便恢复时的地址匹配），按照大于n的最小2的幂次进行页面遍历，更改页的Property状态位为0，并将页首的property值设置为n（以便恢复时进行遍历），返回页首。

- void buddy_free_pages(struct Page* base, size_t n)

遍历查找 rec 数组，当页首地址匹配时获取其对应的偏移量，先通过遍历找偏移量为 offset 的页，再由释放页数 n 再次遍历修改页的使用状态。完成对双向链表处理后，按照buddy system中的释放算法先对 longest=0 的二叉树节点进行修改，再递归合并伙伴节点，更改父节点longest值。最后消除此次的rec记录并减少分配块数的值。

测试样例部分：检测目的等见注释

```
    struct Page *p0, *p1, *p2, *p3, *p4;
    p0 = p1 = p2 = p3 = p4 =NULL;

    assert((p0 = alloc_page()) != NULL);
    //如果alloc_page成功返回一个非空指针,则assert判断为true,测试通过
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(p0 != p1 && p0 != p2 && p1 != p2);//返回的地址不相同
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);
    //确保返回的是空闲页面而不是引用计数器 reference counter不为零（即映射过）的界面
    free_page(p0);
    free_page(p1);
    free_page(p2);

    p1=alloc_pages(500);
    p2=alloc_pages(510);
    cprintf("p1 %p\n",p1);//打印分配后返回的起始地址
    cprintf("p2 %p\n",p2);
    free_pages(p1,250);
    free_pages(p2,510);
    free_pages(p1+250,250);//分两部分返回p1

    p0=alloc_pages(1024);
    cprintf("p0 %p\n",p0);
```

```
26      assert(p0 == p1);//该判断考虑p1分配1024左子树开头页地址,p2分配1024有右子树开头页地
27                       //当两者全部释放后，p0的1024页面返回的地址应该和p1的地址一样
28
29      p1=alloc_pages(69);
30      p2=alloc_pages(36);
31      assert(p1+128==p2);//检查是否相邻，可手绘二叉树看出两者间的地址关系
32
33      cprintf("p1 %p\n",p1);
34      cprintf("p2 %p\n",p2);
35      p3=alloc_pages(88);
36      assert(p1+256==p3);//检查p3有没有和p1重叠
37
38      cprintf("p3 %p\n",p3);
39      free_pages(p1,69);//释放p1
40
41      p4=alloc_pages(63);
42      cprintf("p4 %p\n",p4);
43
44      assert(p2+64==p4);//检查p2，p4是否相邻
45      //因为伙伴系统分配空间时找到两个相符合的节点中内存较小的结点!
46
47      free_pages(p2,36);
48
49      free_pages(p4,63);
50
51      free_pages(p3,88);
52      free_pages(p0,1024);//全部释放
```

测试成功截图：

```
Special kernel symbols:
  entry  0xffffffffc0200036 (virtual)
  etext  0xffffffffc020184a (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0292e78 (virtual)
Kernel executable memory footprint: 588KB
memory management: buddy_system_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
p1 0xffffffffc029c8f8
p2 0xffffffffc02a18f8
p0 0xffffffffc029c8f8
p1 0xffffffffc02a68f8
p2 0xffffffffc02a7cf8
p3 0xffffffffc02a90f8
p4 0xffffffffc02a86f8
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
```

需要注意的是，本 buddy system 的成功实现需要内核付出一定的内存空间来存储 root 数组和 rec 数组，这会改变内存分配的布局，导致在此buddy上应用的 slub 算法报错。经过对该二叉树结构存放位置的思考，我们对原 buddy system 代码进行改进，安排一定量的内存页来存放二叉树记录，安排剩下的内存页做可分配给应用的页。该实现的底层思路与 buddy system 完全一致，具体差别在于存放二叉树记录页的数目确定，见下方更新后的buddy_init_memmap 函数：

```
static void buddy_init_memmap(struct Page *base, size_t n)
{
...
    if (n < 512)
    {
        full_tree_size = POWER_ROUND_UP(n - 1);//大于a的最小的2^k
        record_area_size = 1;
    }//记录一个二叉树节点需要sizeof(size_t)=8字节，如果根节点为256页的话，总共的节点个数为
    //256*2-1=511≈512个，那么这些节点的大小总共为 512*8=4096字节 正好为一页的大小
    //因此，当初始化中给定的页面少于512的话，建立根节点大小为256页的buddy system，并用1页记录二叉树节点。

    else
    {
        full_tree_size = POWER_ROUND_DOWN(n);//小于a的最大的2^k
        record_area_size = full_tree_size * sizeof(size_t) * 2 / PGSIZE;
        if (n > full_tree_size + (record_area_size << 1))
        {
            full_tree_size <<= 1;
            record_area_size <<= 1;
        }//根据内存页数量 n 的大小，动态调整
        //存放二叉树记录的页的数量 record_area_size 和完整二叉树的大小 full_tree_size
    }
    real_tree_size = (full_tree_size < total_size - record_area_size) ? full_tree_size : total_size - record_area_size;
    ...
}
```

经过对 buddy 二叉树结构的页存放调整后，在其上运行的slub算法能成功实现要求，见challenge2。

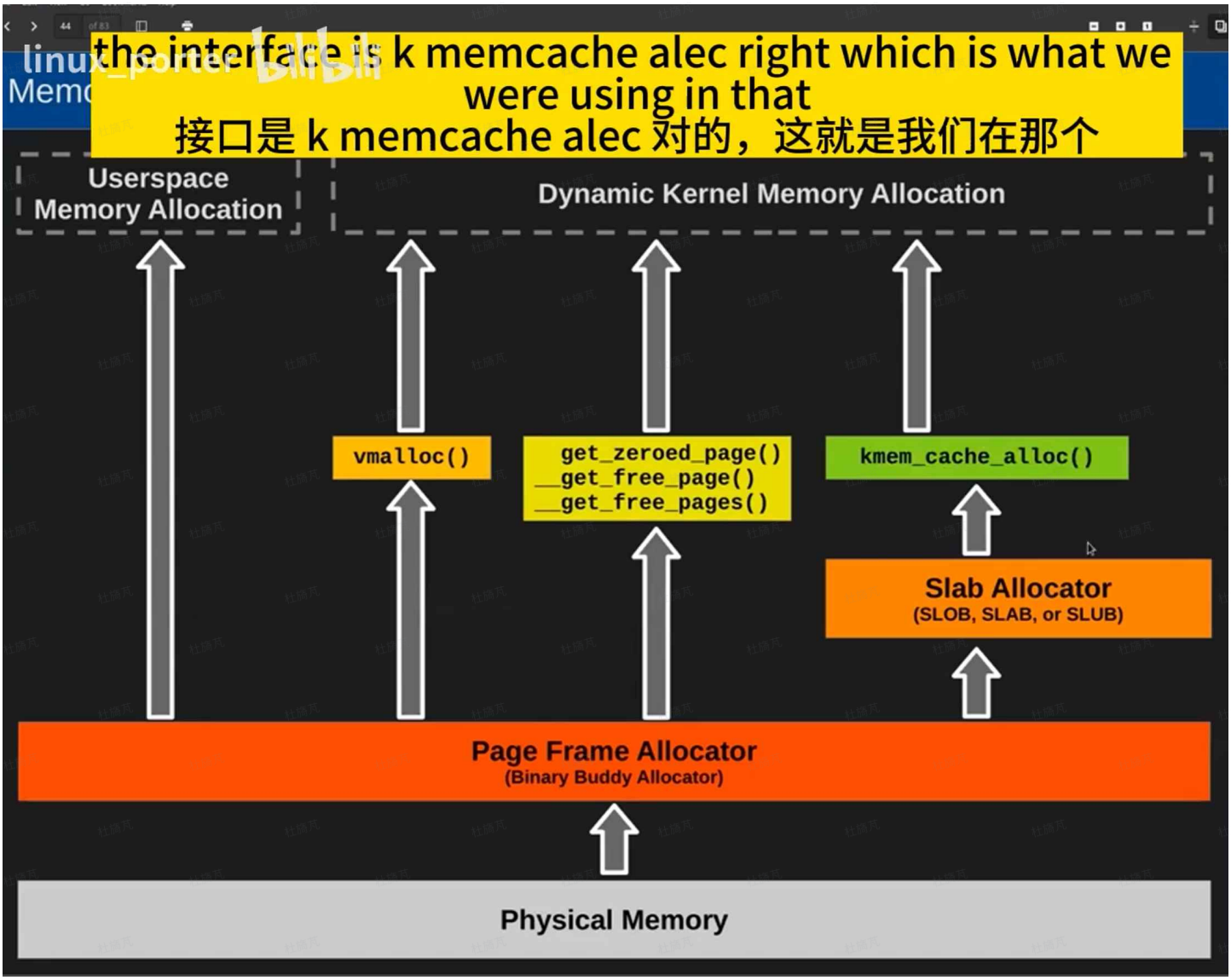
Challenge 2

核心想法：将buddy得到的内存块进一步分块（这样的一块被称为slab内存），只有一种类型的对象将进入slab。这也就是slab allocator，它目前默认由**slub算法**实现。（这些命名有点乱）

由于ucore是一个实验性的OS，所以我们不实现极其复杂的slub算法，下面我们首先介绍经典的Linux slub架构，然后给出我们对slub进行简化的版本。

原本的Linux slub算法

这里先介绍一下Linux内核中经典的slub分配架构（事实上这次实验实现的slub算法相对于下面的这个要简化一些，后面会指出）：

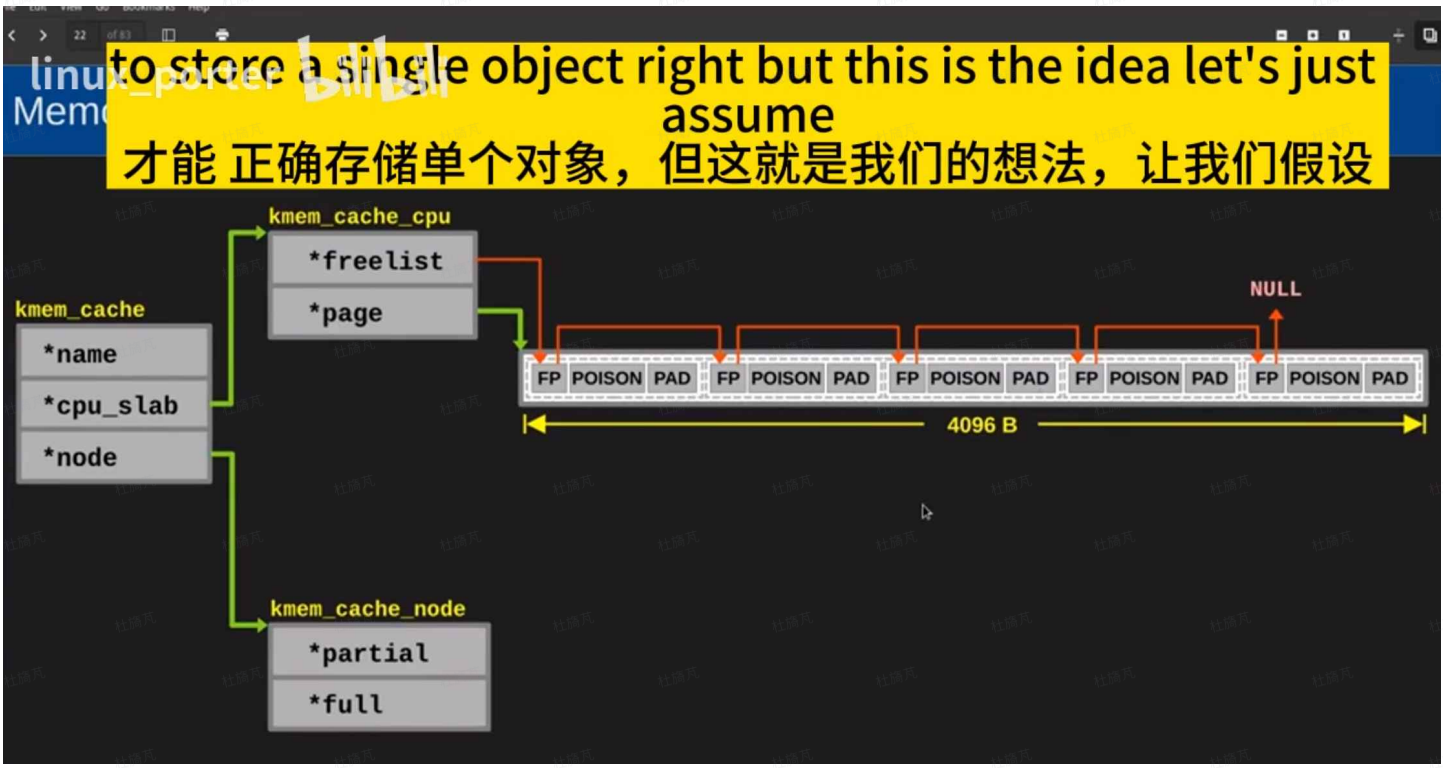


这就是先通过buddy伙伴系统分配页框，然后进一步通过分配器（可以通过slob、slab或者**slub**也就是本次实验需要使用的算法实现）细分到slab内存块，这样就实现了任意大小的内存单元分配算法。

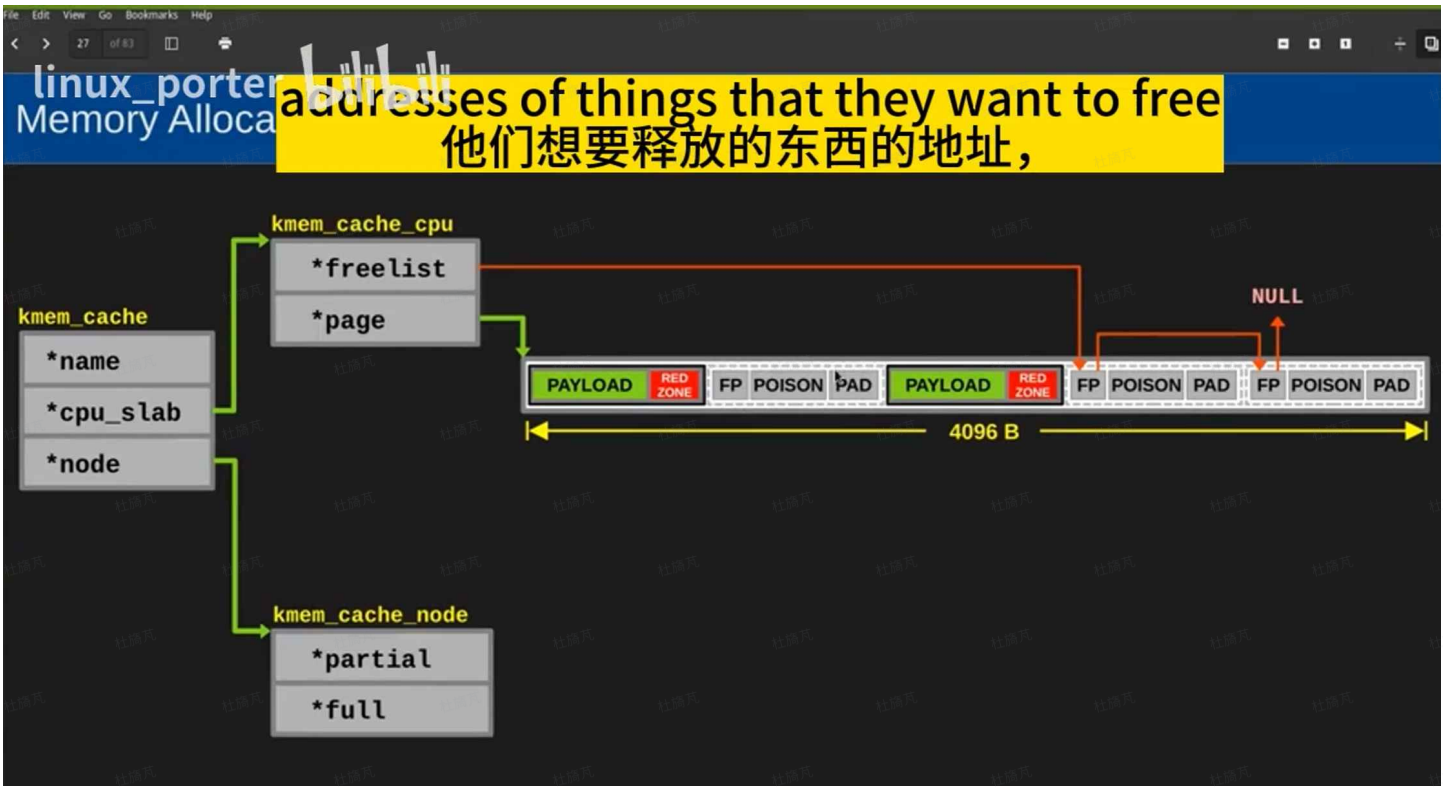
一般来讲，一个分配器具有多个下图中的kmem_cache，不同的kmem_cache对应着不同的chunk size（也就是对象的大小，或者叫更细粒度的内存单元）：

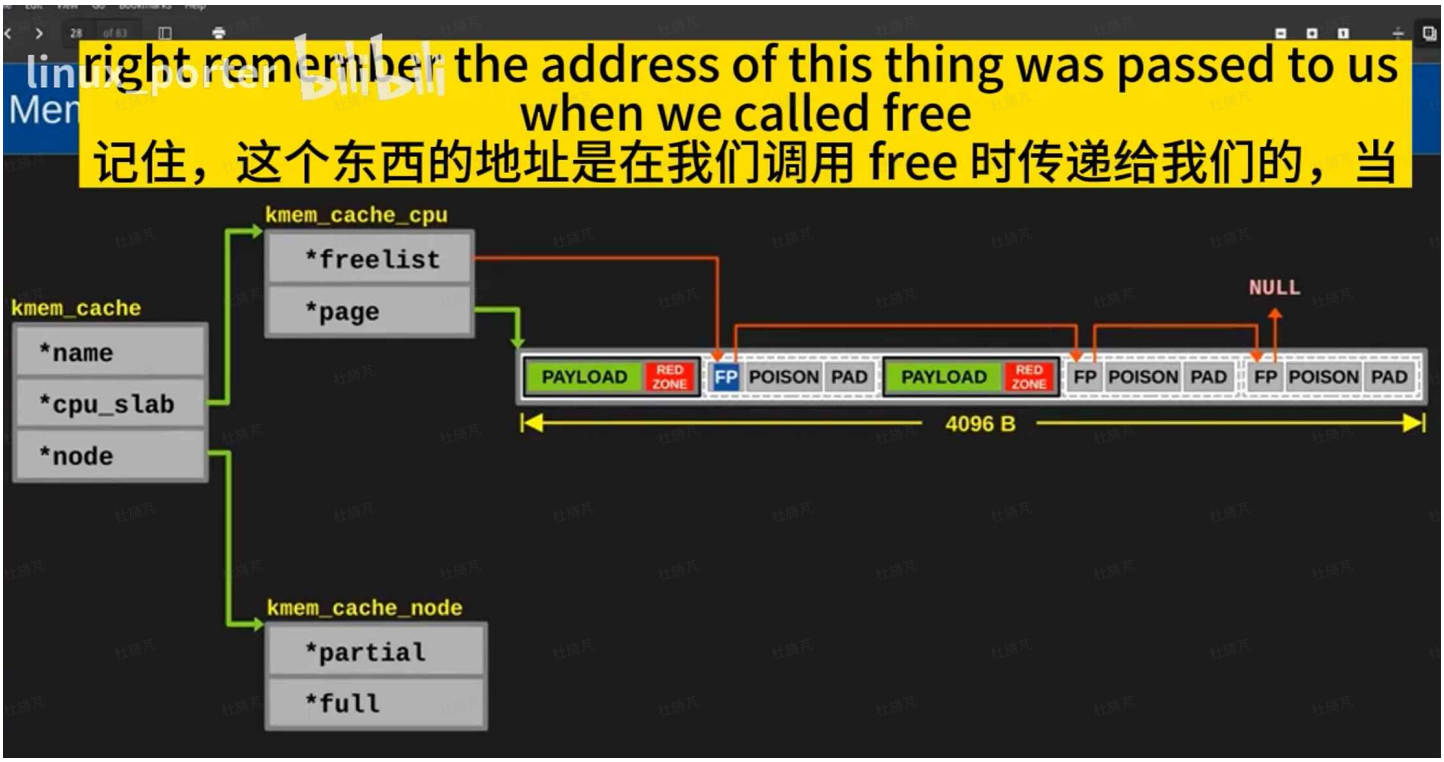
这个结构体有两个指针，指向另外的两个结构体kmem_cache_cpu和kmem_cache_node。每个CPU都有独立的slab，这样计算更加高效，减少了locking，同时因此上面的结构体命名叫做kmem_cache_cpu。

一般来讲，slab的大小是自动识别并由buddy分配的（我们为实现ucore中简易的版本，假定slab的大小为一页大小）：

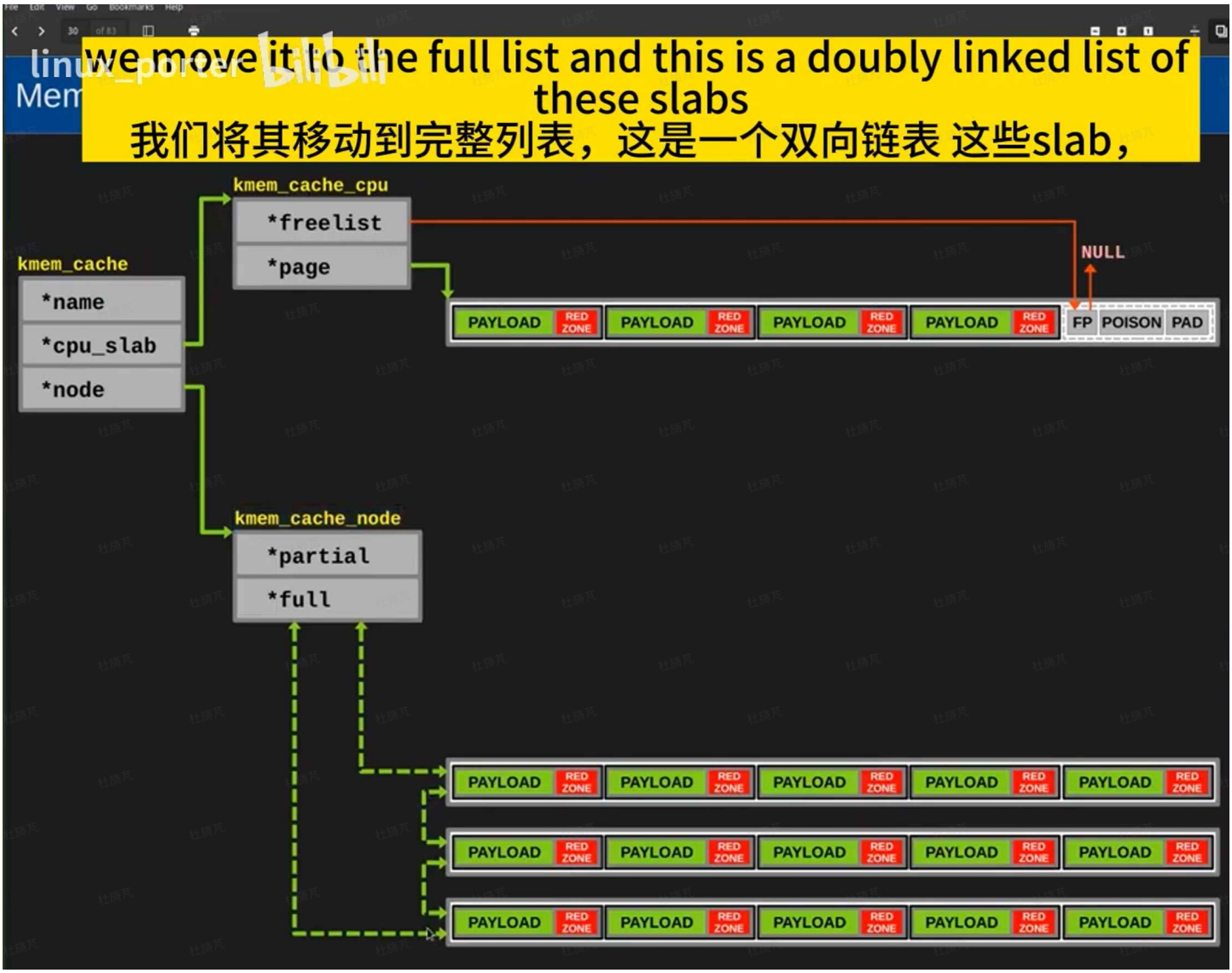


在一个freelist中释放内存，也就类似于空闲链表的更新：

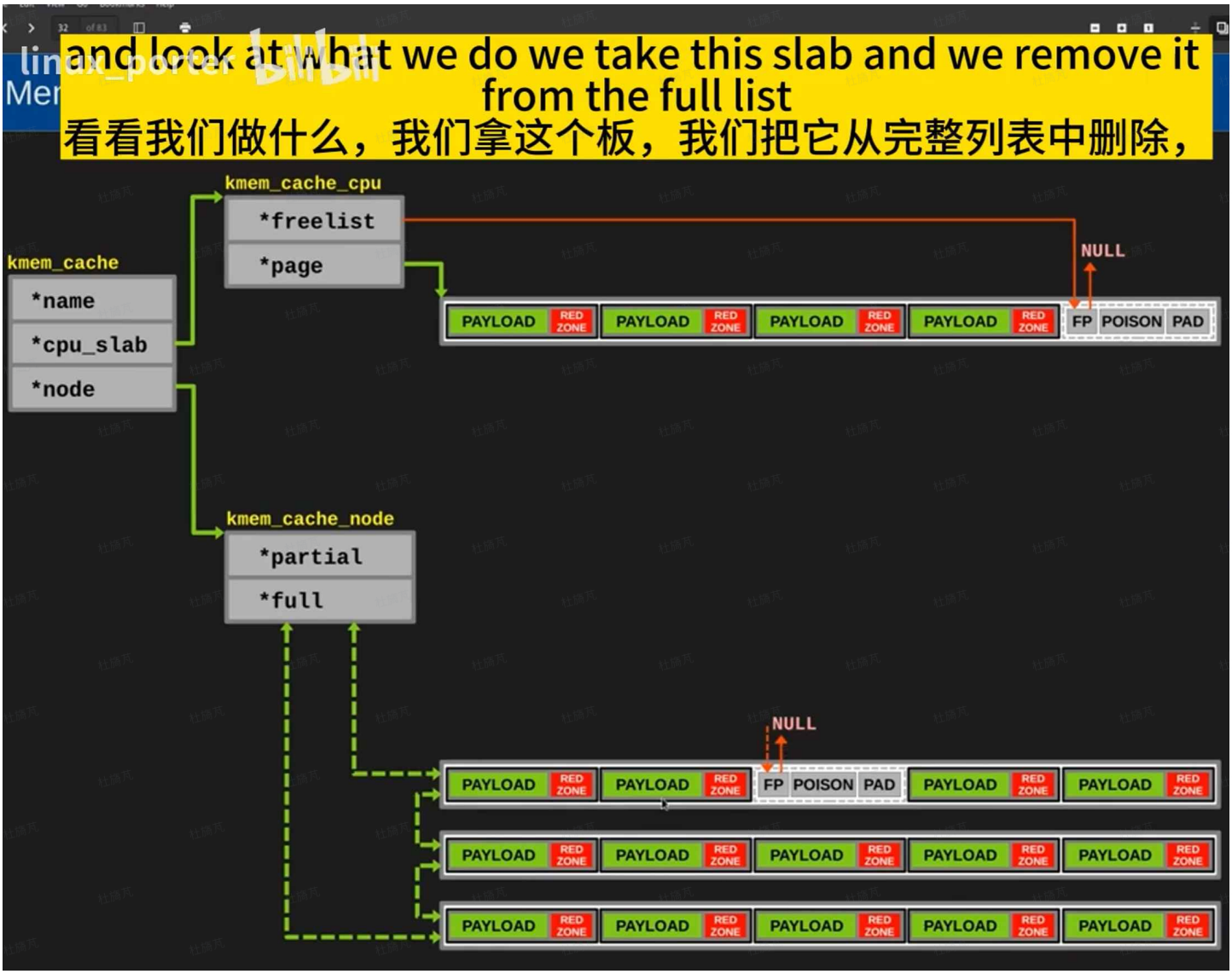


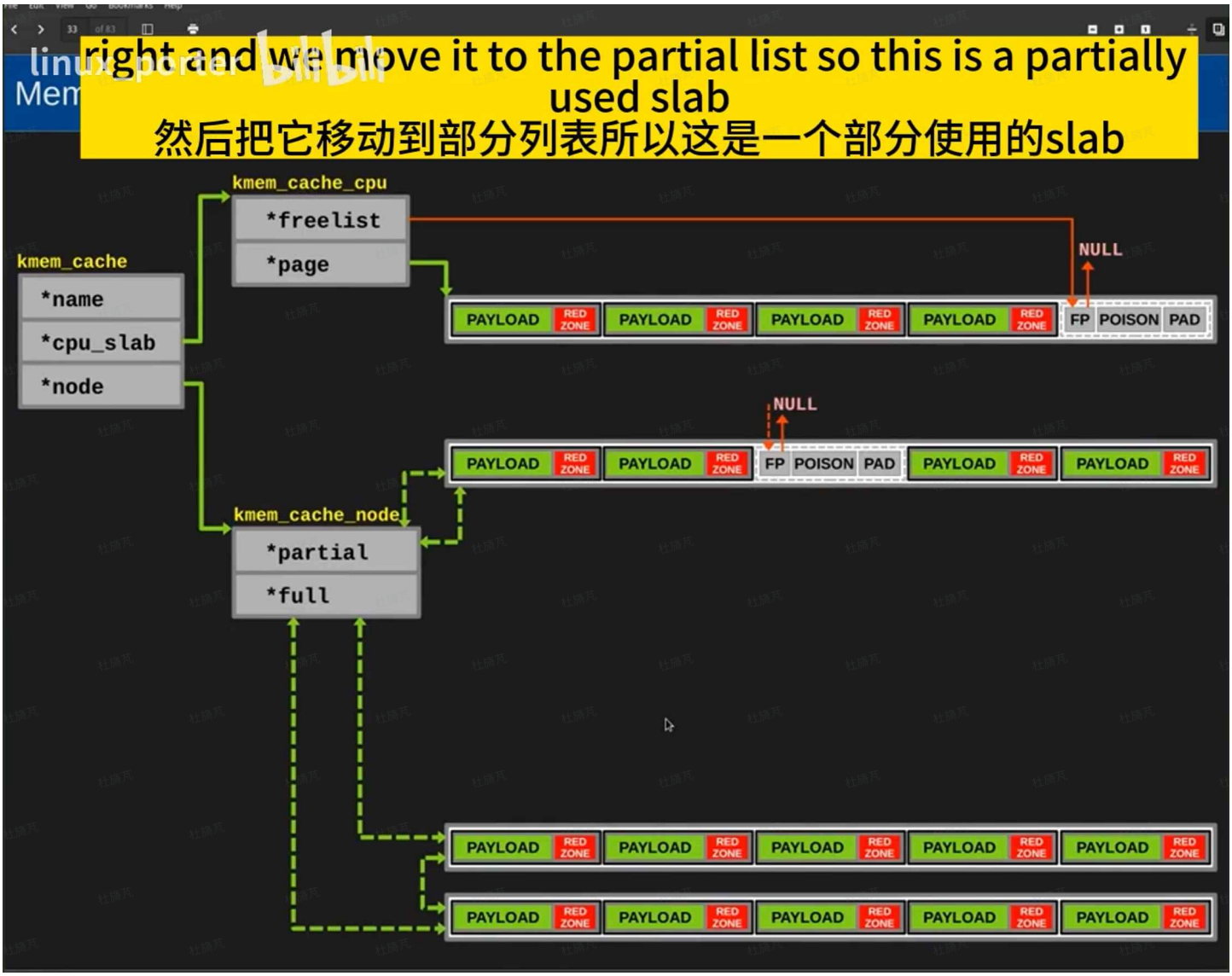


当freelist满了之后会移到full链表中去：



这个时候的释放有些不同，当full链表中的对象被释放的时候，它会被挪到partial（记录部分满的slab内存块）维护的链表中，而不是直接放回freelist中：


and look at what we do we take this slab and we remove it from the full list
看看我们做什么，我们拿这个板，我们把它从完整列表中删除，
kmem_cache_cpu
*freelist
*page
kmem_cache
*name
*cpu_slab
*node
NULL
PAYLOAD
RED ZONE
FP
POISON
PAD
kmem_cache_node
*partial
*full
NULL



这个partial（部分满）就像一个“预备”状态，随时准备交换到上面的freelist中。每次上面的链表全full的话，首先在partial中去找，找不到才从buddy中进行分配。

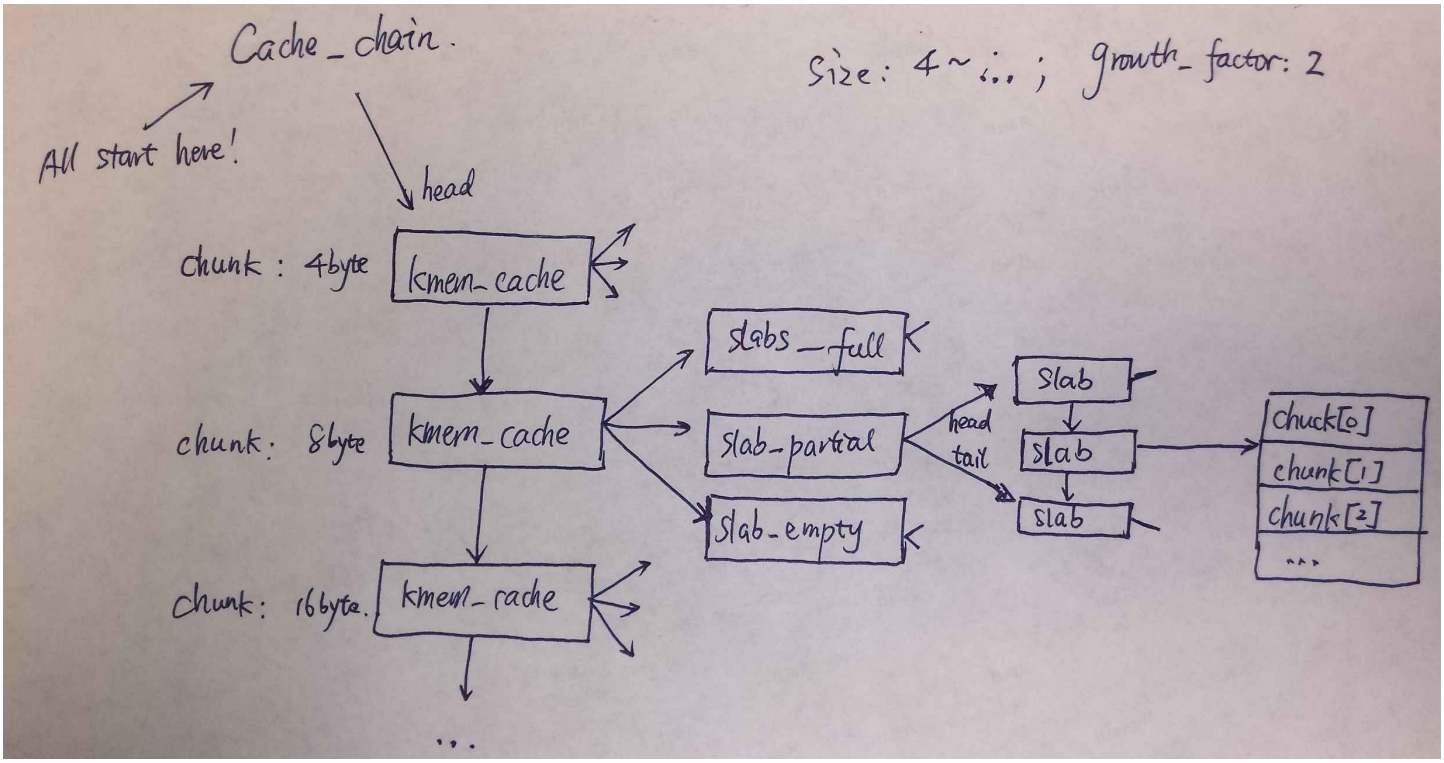
此外，如果有许多partial和freelist中的slab内存空了（不再有对象存放在其中了）的话，可以根据系统内存是否不足决定是否把内存还回系统。

简化的slub分配器

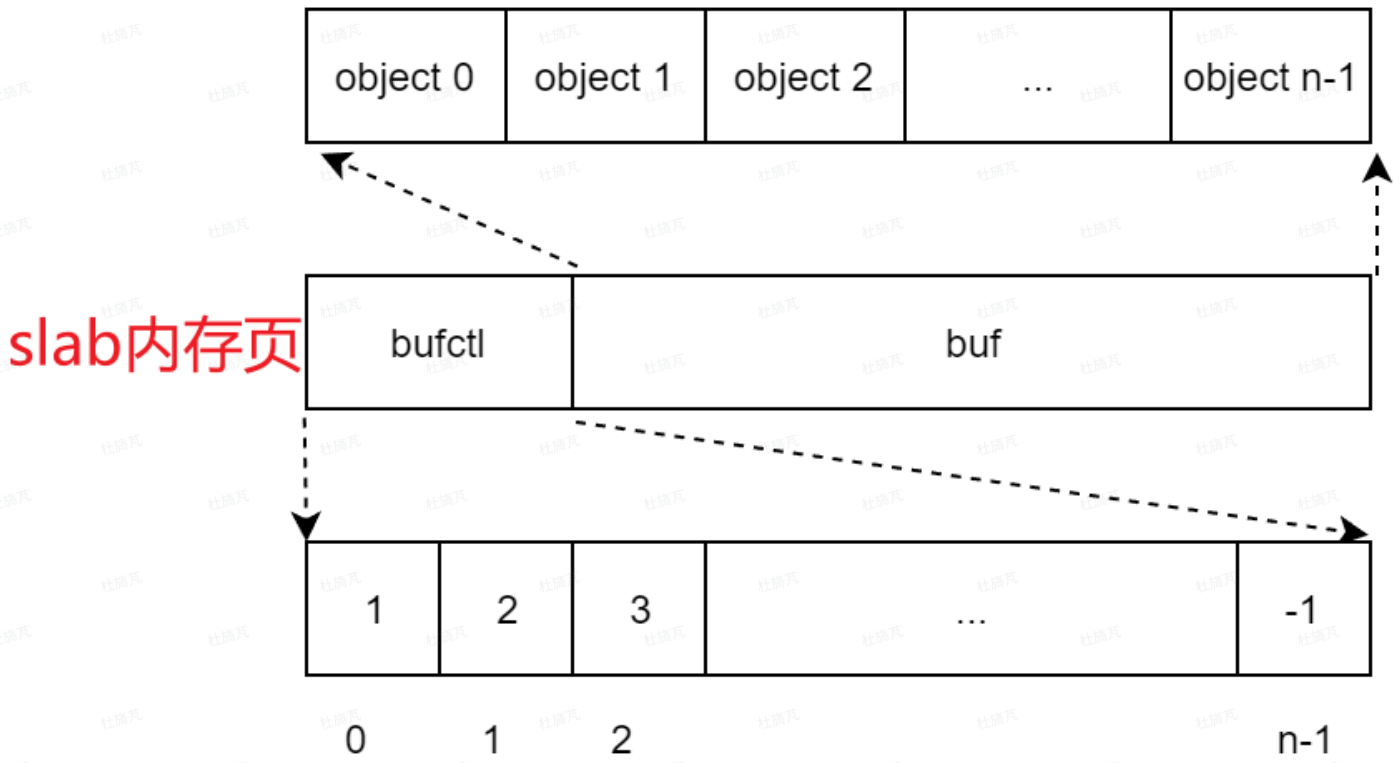
实际上Slub分配算法是非常复杂的，还需要考虑缓存对齐、NUMA等非常多的问题，由于ucore是一个实验性质的操作系统，我们不考虑这些复杂因素了。参考 https://www.kernel.org/doc/gorman/html/understand/understand011.html，**简化的Slub算法结合了Slab算法和Slub算法的部分特征，使用了一些比较有技巧性的实现方法**。如下图所示（下面这个图来自https://github.com/wfgydbu/Slab-Memory-Allocator/tree/master，其中的slab是指那块内存“板子”，与算法名字slub并不冲突），具体的简化为：

- 我们把上面Linux内核中的三个结构体kmem_cache、kmem_cache_cpu和kmem_cache_node合并为了一个结构体kmem_cache（代码中定义为kmem_cache_t），可以称其为**仓库**，该结构体中维护三个链表，**slabs_full**对应分配满对象了的slab内存块，**slab_partial**对应部分被分配的slab内存块，**slab_empty或者叫slab_free**对应完全没被分配过的slab内存块。

- Slab大小固定为一页，对象的大小可以为16, 32, 64, 128, 256, 512, 1024, 2048字节，不允许创建大对象（大于等于一页的对象）仓库。
- 复用Page数据结构，将Slab元数据保存在Page结构体中，使用静态链表的形式保存空闲块的链接记录（下面有图解释）。
- 原slub算法中会自动根据系统内存是否不足决定是否把内存还回系统，这里我们假定slab内存的释放由程序员来手动掌控。



上面的chunk就是对象的大小。slab内存块的复用结构如下：



下面的是一种**静态链表**，记录了当前空闲块的链接信息（下称**空闲表**），由于初始化都是空闲的，所以就是一个从头串到尾的单向链表。上面的buf用于存放实际的对象。

然后稍微剖析一下代码。首先定义两个结构，前文已经介绍了，kmem_cache_t对应kmem_cache的结构体，slab_t对应slab内存块的结构体。

```
// kern/mm/slub.h
struct kmem_cache_t
{
    list_entry_t slabs_full;                                    // 全满Slab链表
    list_entry_t slabs_partial;                                 // 部分空闲Slab链表
    list_entry_t slabs_free;                                    // 全空闲Slab链表
    uint16_t objsize;                                           // 对象大小
    uint16_t num;                                               // 每个Slab保存的对象数目
    void (*ctor)(void *, struct kmem_cache_t *, size_t); // 构造函数
    void (*dtor)(void *, struct kmem_cache_t *, size_t); // 析构函数
    char name[CACHE_NAMELEN];                                   // 仓库名称
    list_entry_t cache_link;                                    // 仓库链表
};

//kern/mm/slub.c
struct slab_t
{
    int ref;                        // 页的引用次数（保留）
    struct kmem_cache_t *cachep; // 仓库对象指针
    uint16_t inuse;                 // 已分配对象的数目
    int16_t free;                   // 下一个空闲对象偏移量，也就是静态链表的头部
    list_entry_t slab_link;      // Slab链表
};
```

构造函数和析构函数是可有可无的，仅在测试的时候使用。其他成员变量的解释在注释中已经比较清楚了。

```
static list_entry_t cache_chain; // cache_chain对应的链表会链接所有kmem_cache
static struct kmem_cache_t cache_cache; // kmem_cache_t仓库，kmem_cache_t也是由Slab算法分配的，因此它会cache所有的kmem_cache_t
static struct kmem_cache_t *sized_caches[SIZED_CACHE_NUM]; // 指针数组，数组的每一个元素指向1个固定大小的内置仓库，共有8个内置仓库，这些仓库存放实际的对象
static char *cache_cache_name = "cache"; // kmem_cache_t仓库的名字
static char *sized_cache_name = "sized"; // 内置仓库的名字
```

由于代码过长，介绍一些关键的函数。首先介绍init.c中会直接调用的kmem_init函数：

```
/* 1. 初始化kmem_cache_t仓库：由于kmem_cache_t也是由Slab算法分配的，所以需要预先手动初
   2. 初始化内置仓库：初始化8个固定大小的内置仓库
   3. 测试slub分配器
*/
void kmem_init()
{

    // 1. 初始化kmem_cache的cache仓库，简称为cache_cache
    cache_cache.objsize = sizeof(struct kmem_cache_t);
    cache_cache.num = PGSIZE / (sizeof(int16_t) + sizeof(struct kmem_cache_t));
    cache_cache.ctor = NULL;
    cache_cache.dtor = NULL;
    memcpy(cache_cache.name, cache_cache_name, CACHE_NAMELEN);
    // 初始化cache_cache的链表
    list_init(&(cache_cache.slabs_full));
    list_init(&(cache_cache.slabs_partial));
    list_init(&(cache_cache.slabs_free));

    // 初始化cache_chain（这个链表会串起来所有的kmem_cache_t）
    list_init(&(cache_chain));
    // 将第一个kmem_cache_t也就是cache_cache加入到cache_chain对应的链表中
    list_add(&(cache_chain), &(cache_cache.cache_link));

    // 2. 初始化8个固定大小的内置仓库
    for (int i = 0, size = 16; i < SIZED_CACHE_NUM; i++, size *= 2)
        sized_caches[i] = kmem_cache_create(sized_cache_name, size, NULL, NULL);

    // 3. 进行测试
    check_kmem();
}
```

这个函数的功能就是初始化slub分配器，它有一个全局的存放所有kmem_cache的cache仓库（有点套娃的感觉，所以叫cache_cache，这个cache_cache仓库的对象类型就是kmem_cache_t），还有8个固定大小的内置仓库，它们分别存放的就是16, 32, 64, 128, 256, 512, 1024, 2048这8种字节大小的对象。

此函数中调用的kmem_cache_create函数作用为在cache_cache中创建一个kmem_cache：

```
/* 创建一个kmem_cache
   从cache_cache中获得一个对象，初始化成员，最后将对象加入仓库链表cache_chain
*/
struct kmem_cache_t *
kmem_cache_create(const char *name, size_t size,
                  void (*ctor)(void *, struct kmem_cache_t *, size_t),
                  void (*dtor)(void *, struct kmem_cache_t *, size_t))
```

```
{
    // 至少得装得下一个对象，2是空闲表一项的大小
    assert(size <= (PGSIZE - 2));
    // sized_cache是从cache_cache中分配出来的
    struct kmem_cache_t *cachep = kmem_cache_alloc(&(cache_cache));
    if (cachep != NULL)
    {
        cachep->objsize = size;
        // 计算Slab中对象的数目，由于空闲表每一项占用2字节，所以每个Slab的对象数目就是：4096字节/(2字节+对象大小)
        cachep->num = PGSIZE / (sizeof(int16_t) + size);
        cachep->ctor = ctor;
        cachep->dtor = dtor;
        // 初始化名字
        memcpy(cachep->name, name, CACHE_NAMELEN);
        // 初始化链表
        list_init(&(cachep->slabs_full));
        list_init(&(cachep->slabs_partial));
        list_init(&(cachep->slabs_free));
        // 加入cache_chain中
        list_add(&(cache_chain), &(cachep->cache_link));
    }
    return cachep;
}
```

可以看出，sized_cache是从cache_cache中分配出来的。逻辑已经很清晰地写在了代码注释中，我们进一步看看kmem_cache_alloc函数：

```
/* 从cachep指向的仓库中分配一个对象
   1. 先查找slabs_partial（部分满），如果没找到空闲区域则查找slabs_free（全空）
   2. 还是没找到就从buddy中申请一个新的slab
   3. 分配内存，并更新各个链表与记录
   4. 从slab分配一个对象后，如果slab变满，那么将slab加入slabs_full
*/
void *
kmem_cache_alloc(struct kmem_cache_t *cachep)
{
    list_entry_t *le = NULL;
    // 1. 先查找slabs_partial
    if (!list_empty(&(cachep->slabs_partial)))
        le = list_next(&(cachep->slabs_partial));
    // 再查找slabs_free
    else
    {
        // 2. 还是没找到就从buddy中申请一个新的slab，对应于kmem_cache_grow函数
```

```
        if (list_empty(&(cachep->slabs_free)) && kmem_cache_grow(cachep) == NULL
            // 实在是分配不了了
            return NULL;
        // 申请之后slabs_free就不是空了，所以是能取出来的
        le = list_next(&(cachep->slabs_free));
    }
    // 3. 分配内存，并更新各个链表与记录
    // 先把这一块slab摘下来
    list_del(le);
    // 把这个list_entry转为page，再转为slab块
    struct slab_t *slab = le2slab(le, page_link);
    // 转为虚拟地址
    void *kva = slab2kva(slab);
    int16_t *bufctl = kva;
    // 偏移到buf
    void *buf = bufctl + cachep->num;
    // 从buf根据free偏移到第一个还能申请来存放对象的地址
    void *objp = buf + slab->free * cachep->objsize;
    // 取下这片区域，标记为使用，更新slab的空闲表
    slab->inuse++;
    slab->free = bufctl[slab->free];
    // 4. 从slab分配一个对象后，如果slab变满，那么将slab加入slabs_full
    if (slab->inuse == cachep->num)
        list_add(&(cachep->slabs_full), le);
    else
        // 否则就放入slabs_partial
        list_add(&(cachep->slabs_partial), le);
    return objp;
}
```

这个函数的主要作用是从cachep指向的仓库中分配一块内存给对象使用。由于仓库有三个链表，因此结合它们的性质，寻找链表的步骤如下:

1. 先查找slabs_partial（部分满），如果没找到空闲区域则查找slabs_free（全空）
2. 还是没找到，说明该仓库目前内存不足，从buddy中申请一个新的slab
3. 分配内存，并更新各个链表与记录
4. 从slab分配一个对象后，如果slab变满，那么将slab加入slabs_full，否则放入slabs_partial

我们再来看看kmem_cache_grow函数：

```
/* 从buddy中申请一页内存作为完全空闲的slab
   1. 从buddy中申请一个page
   2. 初始化空闲链表bufctl，初始化Slab元数据
   3. 初始化buf中的对象
```

```
    4. 最后将新的Slab加入到仓库的slab_free中
*/
static void *
kmem_cache_grow(struct kmem_cache_t *cachep)
{
    // 1. 从buddy中申请一个page
    struct Page *page = alloc_page(); // ! 这个是manager中的函数，对接上buddy的接口
    // 获取页的虚拟地址，这里需要自己实现
    void *kva = page2kva(page);
    // 2. 初始化Slab元数据，初始化空闲链表bufctl
    struct slab_t *slab = (struct slab_t *)page;
    slab->cachep = cachep;          // 设置仓库对象
    slab->inuse = slab->free = 0; // 当前没有分配过
    // 在这一页开头构建静态链表（由于是初始化，直接按顺序穿起来），记录可用内存区块的偏移
    int16_t *bufctl = kva;
    for (int i = 1; i < cachep->num; i++)
        bufctl[i - 1] = i;
    bufctl[cachep->num - 1] = -1;
    // 3. 初始化buf中的对象
    // 计算静态链表后的虚拟地址
    void *buf = bufctl + cachep->num;
    // 有构造函数才做
    if (cachep->ctor)
        for (void *p = buf; p < buf + cachep->objsize * cachep->num; p += cachep
            cachep->ctor(p, cachep, cachep->objsize);
    // 4. 最后将新的Slab加入到仓库的slab_free中
    list_add(&(cachep->slabs_free), &(slab->slab_link)); // 当前这一页全空闲
    return slab;
}
```

思路也是非常清晰的，调用alloc_page()接口（可以当作pmm_manager的成员函数来想），然后通过虚拟地址访问这个page，初始化其中的空闲表（静态链表），然后用构造函数初始化其中的对象，最后将新的slab加入到仓库的slab_free链表中。

这样，slub分配器的基本框架就出来了，还有一些释放相关的函数，先对kmem_slab_destroy进行介绍：

```
// 释放cachep中的一个slab块对应的页，析构buf中的对象后将page归还
static void
kmem_slab_destroy(struct kmem_cache_t *cachep, struct slab_t *slab)
{
    // 计算buf的虚拟地址
    struct Page *page = (struct Page *)slab;
    int16_t *bufctl = page2kva(page);
    void *buf = bufctl + cachep->num;
```

```
    // 有析构函数才做
    if (cachep->dtor)
        // 析构每个对象
        for (void *p = buf; p < buf + cachep->objsize * cachep->num; p += cachep
            cachep->dtor(p, cachep, cachep->objsize);
    // 清除property和flag
    page->property = page->flags = 0;
    list_del(&(page->page_link));
    free_page(page); // ! 这个是manager中的函数，对接上buddy的接口
}
```

思路很直接，找到这一页对应的虚拟地址，然后调用析构函数，最后将页清除（property和flag，然后调用buddy提供的free_page接口）。

还有更细粒度的释放函数，也就是释放对象：

```
/* 将对象objp从cachep中的Slab中释放
   也就是将对象空间加入空闲链表（slabs_partial或者slabs_free），更新Slab元信息
   如果Slab变空，那么将Slab加入slabs_free链表
*/
void kmem_cache_free(struct kmem_cache_t *cachep, void *objp)
{
    // 获取对象所在的slab
    void *base = page2kva(pages);
    void *kva = ROUNDDOWN(objp, PGSIZE);
    struct slab_t *slab = (struct slab_t *)&pages[(kva - base) / PGSIZE];
    // 获取对象在slab中的偏移
    int16_t *bufctl = kva;
    void *buf = bufctl + cachep->num;
    int offset = (objp - buf) / cachep->objsize;
    // 更新Slab元信息
    list_del(&(slab->slab_link)); // 在链表中删除
    bufctl[offset] = slab->free;
    slab->inuse--;
    slab->free = offset;
    // 如果Slab变空，那么将Slab加入slabs_free链表，否则加入slabs_partial
    if (slab->inuse == 0)
        list_add(&(cachep->slabs_free), &(slab->slab_link));
    else
        list_add(&(cachep->slabs_partial), &(slab->slab_link));
}
```

需要注意的是释放后链表的更新。

其他的释放内存函数此处由于篇幅过长，就不再讲述了，思路都是相通的。

最后，介绍一下编写的测试样例，实际上slub分配器测试执行的流程是kern_init->kmem_init->check_kmem，也就是说kmem_init函数除了初始化kmem_cache_t仓库和内置仓库，还会对slub分配器进行测试。

check_kmem函数内容如下，详细思路见注释：

```
static void
check_kmem()
{

    assert(sizeof(struct Page) == sizeof(struct slab_t)); // 页的大小等于slab的大小

    size_t fp = nr_free_pages();

    // 创建一个测试仓库，初始化对象内存空间全为TEST_OBJECT_CTVAL
    // test_object大小为2046字节
    struct kmem_cache_t *cp0 = kmem_cache_create(test_object_name, sizeof(struct
    assert(cp0 != NULL);                                                    // 创建成功
    assert(kmem_cache_size(cp0) == sizeof(struct test_object));  // 对象大小一致
    assert(strcmp(kmem_cache_name(cp0), test_object_name) == 0); // 名字一样

    struct test_object *p0, *p1, *p2, *p3, *p4, *p5;
    char *p;
    // 在仓库中分配5个对象
    assert((p0 = kmem_cache_alloc(cp0)) != NULL);
    assert((p1 = kmem_cache_alloc(cp0)) != NULL);
    assert((p2 = kmem_cache_alloc(cp0)) != NULL);
    assert((p3 = kmem_cache_alloc(cp0)) != NULL);
    assert((p4 = kmem_cache_alloc(cp0)) != NULL);
    p = (char *)p4;
    // 对象的初始值应该都是TEST_OBJECT_CTVAL
    for (int i = 0; i < sizeof(struct test_object); i++)
        assert(p[i] == TEST_OBJECT_CTVAL);
    assert((p5 = kmem_cache_zalloc(cp0)) != NULL);
    p = (char *)p5;
    // 由于调用的是zalloc，也就是说分配一个对象之后对象内存区域全初始化为零，所以有p[i] =
    for (int i = 0; i < sizeof(struct test_object); i++)
        assert(p[i] == 0);
    // 由于刚刚总共分配了6个对象（2046+2），现在应该从buddy取出了3页，因此空闲页数比之前
    assert(nr_free_pages() + 3 == fp);
    // 当前3页应当全被占满，都在slabs_full链表中
    assert(list_empty(&(cp0->slabs_free)));
    assert(list_empty(&(cp0->slabs_partial)));
    assert(list_length(&(cp0->slabs_full)) == 3);
    // 释放三个对象，现在slabs_free、slabs_partial、slabs_full应当各有一块slab
    kmem_cache_free(cp0, p3);
```

```
    kmem_cache_free(cp0, p4);
    kmem_cache_free(cp0, p5);
    assert(list_length(&(cp0->slabs_free)) == 1);
    assert(list_length(&(cp0->slabs_partial)) == 1);
    assert(list_length(&(cp0->slabs_full)) == 1);
    // 释放slabs_free链表中的所有slab
    assert(kmem_cache_shrink(cp0) == 1);     // 一共释放了1个slab
    assert(nr_free_pages() + 2 == fp);       // 现在多了一个空闲页
    assert(list_empty(&(cp0->slabs_free))); // slabs_free变空
    // p4处的内存被释放，现在对象内存处的值都是析构函数中的TEST_OBJECT_DTVAL
    p = (char *)p4;
    for (int i = 0; i < sizeof(struct test_object); i++)
        assert(p[i] == TEST_OBJECT_DTVAL);
    // 释放对象，现在slabs_free又多了两块slab
    kmem_cache_free(cp0, p0);
    kmem_cache_free(cp0, p1);
    kmem_cache_free(cp0, p2);
    // 释放所有全空闲slab，共释放2个
    assert(kmem_cache_reap() == 2);
    // 现在页全部空闲了
    assert(nr_free_pages() == fp);
    // 释放仓库
    kmem_cache_destroy(cp0);

    // 在内置仓库中申请内存
    assert((p0 = kmalloc(2048)) != NULL);
    // 空闲页少1
    assert(nr_free_pages() + 1 == fp);
    // 在内置仓库中释放对象
    kfree(p0);
    // 释放后，多出一个全空闲slab，释放掉
    assert(kmem_cache_reap() == 1);
    // 空闲页复原
    assert(nr_free_pages() == fp);

    cprintf("check_kmem() succeeded!\n");
}
```

如下图，程序正常执行，并且通过测试：

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS 1

  etext  0xffffffffc0202a9c (virtual)
  edata  0xffffffffc0207010 (virtual)
  end    0xffffffffc0207560 (virtual)
Kernel executable memory footprint: 30KB
memory management: buddy_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
freemem = 0x80348000
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
check_kmem() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
```

Challenge3

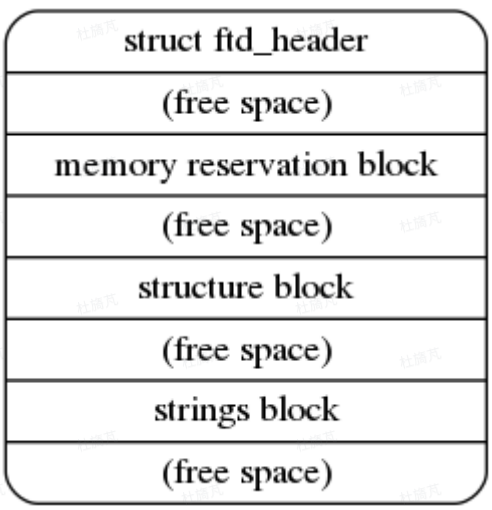
首先通过查看OpenSBI的文档可知，OpenSBI 在进入内核前会将扁平化的设备树位置写入内存中，并且将设备树对应的内存物理地址存入 `a1` 寄存器（第二个参数）中，于是我使用gdb调试工具去找到了设备树的内存物理地址，如下：(这里需要先修改entry.S文件，请参考报告开头的**补充**)

```
(gdb) b *0x80200000
Breakpoint 1 at 0x80200000
(gdb) c
Continuing.

Breakpoint 1, 0x0000000080200000 in ?? ()
(gdb) info r $a1
a1             0x0000000082200000       2183135232
(gdb)
```

设备树格式

这里通过查阅设备树资料，可以了解到设备树的格式如下：



在设备树头部结构体中，会存储多种信息，例如魔数、大小、不同块的偏移地址等等。这里我们需要获取到可用的物理内存空间，同样查阅资料了解到，在structure block中会存在不少的节点，其中我们只需要找到memory节点中的信息即可实现我们的目标，在这个结构体的内部，结构是通过一些用数值象征的标识符来进行划分的，标识符如下：

1. `FDT_BEGIN_NODE` 标识符后跟随着一个字符串，表示节点的名称。名称后对齐到四字节。
2. `FDT_END_NODE` 标识符后没有附加信息，表示节点的结束。
3. `FDT_PROP`：表示一个属性，标识符后跟随着一个结构体，包括这一属性值的长度以及属性名称在 strings block 中的偏移量。之后跟随着属性值，属性值之后会通过填充 0 对齐到 4 字节。
4. `FDT_NOP`：表示一个空节点，解析时直接跳过即可
5. `FDT_END`：表示设备树的结束。

```
#define FDT_BEGIN_NODE      0x01000000
#define FDT_END_NODE        0x02000000
#define FDT_PROP            0x03000000
#define FDT_NOP             0x04000000
#define FDT_END             0x09000000
```

(注：上图是大端字节序)

代码设计思路

主要的思路就是通过解析设备树来实现，根据设备树的结构进行设备树的解析，然后匹配到其中的memory节点，然后将其中信息进行输出即可。

首先gdb调试看一下设备树，如下：

```
Breakpoint 1, 0x0000000080200000 in ?? ()
(gdb) info r a1
a1             0x0000000082200000       2183135232
(gdb) x/40x $a1
0x82200000:     0xedfe0dd0      0x260d0000      0x38000000      0xb00b0000
0x82200010:     0x28000000      0x11000000      0x02000000      0x00000000
0x82200020:     0x76010000      0x780b0000      0x00000000      0x00000000
0x82200030:     0x00000000      0x00000000      0x01000000      0x00000000
0x82200040:     0x03000000      0x04000000      0x1d000000      0x02000000
0x82200050:     0x03000000      0x04000000      0x11000000      0x02000000
0x82200060:     0x03000000      0x0d000000      0x06000000      0x63736972
0x82200070:     0x69762d76      0x6f697472      0x6d657100      0x03000000
0x82200080:     0x12000000      0x00000000      0x63736972      0x69762d76
0x82200090:     0x6f697472      0x6d65712c      0x00000075      0x01000000
(gdb)
```

可以看到都是大端字节序存储信息，于是在解析的时候需要编写一个大小端字节序转换的函数，如下：

```
1 uint32_t my_bswap32(uint32_t x) {
```

```
    return ((x & 0xFF000000) >> 24) |
           ((x & 0x00FF0000) >> 8)  |
           ((x & 0x0000FF00) << 8)  |
           ((x & 0x000000FF) << 24);
}
```

然后编写设备树搜寻函数搜寻到memory节点并进行输出，其中函数利用的参数是设备树的物理内存地址，即我们上文调试到的0x0000000082200000。于是在init.c中调用函数并传参：

```
    fdt_header_t* fdt_header = (fdt_header_t*)(0x0000000082200000);//gdb's resul
    cprintf("\n \nChallenge3-check: \nfdt_magic:0x%08x\n",
            my_bswap32(fdt_header->magic));
    select_dtb(fdt_header);
```

（特别感谢：感谢好朋友梅神的解答，帮我解决了不少stain --From 冯思程）

测试

具体的select_dtb函数请看GitHub的代码链接，最终make qemu得到的输出结果如下：

```
Challenge3-check:
fdt_magic:0xd00dfeed
memory@80000000 {
device_type: 6d656d6f 727900
reg: 00000000 80000000 00000000 08000000
}
```

可以看到成功的搜索到了memory节点，下面对其中内容进行分析：

1. `device_type` 这是一个字符串，转换后就是代表memory，表示节点信息。
2. `reg` 这其中存了两个64位的字段，第一个字段是0x0000000080000000，为可用物理内存空间的起始地址；第二个字段是0x0000000008000000，表示空间大小，即128MiB，这个大小为QEMU默认分配的大小。

输出结果正确无误，说明完成了challenge3的内容。

实验知识点

在本实验中，重要的知识点包括：

1. 虚拟页表的建立和使用：这涉及操作系统中的内存管理原理。操作系统通过虚拟内存技术将进程使用的虚拟地址空间映射到物理内存中。它使得多个进程可以共享有限的物理内存，并且可以更高效地管理内存资源。
2. 物理内存管理：这也是操作系统中的核心概念。操作系统负责管理计算机的物理内存，以便为不同的进程提供足够的内存空间。它包括分配、回收和调度等方面的工作。
3. RISC-V 中的 SV39 页表机制：RISC-V 是一个开源指令集架构，它定义了一系列的特权级别和页表格式来管理内存。SV39 页表机制是 RISC-V 中的一种，它规定了 39 位的虚拟地址空间。SV39 将虚拟地址空间分成三级页表（页全局目录、页中间目录、页表），并且在页表项中存储了物理页框的信息以进行映射。
4. x86 系统的页表管理机制：x86 架构也有自己的页表管理机制。它将虚拟地址空间映射到物理地址空间，但其结构和管理方式与 RISC-V 中的 SV39 页表机制略有不同。

这些知识点之间的关系是，虚拟页表的建立和使用是内存管理的重要组成部分，而RISC-V 中的 SV39 页表机制是其中一种实现方式。物理内存管理是确保操作系统能够高效地利用可用的物理内存。x86 系统的页表管理机制和RISC-V 中的 SV39 页表机制有一些相似之处，但也有不同的设计理念和实现细节。

实验中没有涉及 SV48、SV57 等 RISC-V 的其他页表机制，这是因为 SV39 已经能够满足实验的需求。同样地，尽管在理论课上讨论了 x86 系统的页表管理机制，但没有选择在本次实验中实现相关内容。