Lab2实验报告

练习一

先进行物理内存布局的探测，得到内存映射地址（bootloader）。物理内存以页的形式进行管理，页的信息保存在Page结构中，而Page以链式结构保存在链表free_area_t中。通过定义pmm_manager实现对物理内存的管理，包含了初始化需要管理的物理内存空间，初始化链表，内存空间分配释放等功能的函数。在内核启动后，会初始化pmm_manager，调用page_init，使用init_memmap将内核映射地址中的物理内存纳入物理内存页管理（将空闲页初始化并加入链表），接下来就可以使用pmm_manager中的空间分配和释放等函数进行内存管理了。

default_init:

```
default_init(void) {
    list_init(&free_list);// 初始化链表头节点
    nr_free = 0;//nr_free可以理解为在这里可以使用的一个全局变量，记录可用的物理页面数
}
```

其中free_list定义在#define free_list (free_area.free_list)

而free_area是free_area_t 的实例化，free_area_t定义在memlayout.h文件

```
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // 空闲页个数
} free_area_t;
```

初始条件时，空闲物理页可能都是连续的，随着物理页的分配与释放，大的连续内存空闲块会被分割为地址不连续的多个小连续内存空闲块。因此使用一个双向链表进行管理，定义free_area_t数据结构，包含一个list_entry结构的双向链表指针，记录当前空闲页个数的无符号整型，包括一个指向空闲页面链表的头节点的指针（free_list），以及一个表示当前空闲页面数量的计数器（nr_free），定义了两个成员：

unsigned int nr_free;
这是一个名为 nr_free 的成员，其类型为 unsigned int。这个成员用于存储当前在这个空闲列表中空闲页面的数量。unsigned int 是一个无符号整数类型，意味着它只能存储非负整数值。

list_entry_t free_list;
这是一个名为 free_list 的成员，其类型为 list_entry_t。list_entry_t 定义在libs/list.h文件中

```
struct list_entry {
    struct list_entry *prev, *next;
};
typedef struct list_entry list_entry_t;
```

结构体 `list_entry` 定义了一个双向链表节点的基本结构。在双向链表中，每个节点都包含指向它前一个节点和后一个节点的指针，从而允许从链表中的任何节点向前或向后遍历。

具体来说，`list_entry` 结构体包含两个成员：

1. `struct list_entry *prev;`

- 指向当前节点前一个节点的指针。如果当前节点是链表的第一个节点，则这个指针通常被设置为 `NULL`，表示没有前一个节点。

2. `struct list_entry *next;`

   - 指向当前节点后一个节点的指针。如果当前节点是链表的最后一个节点，则这个指针通常也被设置为 `NULL`，表示没有后一个节点。

结构体中的 `struct list_entry` 类型表示这是一个递归定义，即每个 `list_entry` 结构体都包含指向其他 `list_entry` 结构体的指针。

总体而言，default_init初始化了一个双向带有头节点且为空的链表，表示当前系统没有空闲页面。

default_init_memmap:

kern_init（内核初始化） --> pmm_init（物理内存初始化）-->page_init（物理地址的初始化过程）-->init_memmap

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);//确保传入的页面数量 n 大于 0
    struct Page *p = base;//指针 p 从 base 开始遍历所有的 Page 结构体
    for (; p != base + n; p ++) {
        assert(PageReserved(p));//检查是否被保留，即初始化前未被使用或分配
        p->flags = p->property = 0;//页面的标志和属性为0
        set_page_ref(p, 0);//函数将页面的引用计数设置为 0，即没有被引用或分配
    }
    base->property = n;//内存映射区域包含 n 个页面
    SetPageProperty(base);//起始页面的属性设置为页面数量，将该页面设置为已分配
    nr_free += n;//增加全局或局部的空闲页面计数器 nr_free
    if (list_empty(&free_list)) {//检查列表是否为空
        list_add(&free_list, &(base->page_link));//添加到空闲列表的头部
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {//遍历空闲列表
            struct Page* page = le2page(le, page_link);
            if (base < page) {//找到属于其的位置
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}

//#define SetPageProperty(page)       set_bit(PG_property, &((page)->flags))
```

用于初始化一个物理内存映射区域，该区域由 `n` 个连续的 `Page` 结构体组成，这些结构体从 `base` 指针开始。会将给定的一段页面地址范围设置为不可用，并将其加入到空闲页面链表中，便于后续的内存分配，即管理和初始化新的一块可用内存块。

`Page` 结构体定义在memlayout.h文件

```
struct Page {
    int ref;                        // 页面的引用计数器
    uint64_t flags;                 // 64位的标志数组，描述页面的状态
    unsigned int property;          // 头页（连续空闲块地址最小的一页）记录空闲块
    list_entry_t page_link;         //  多个连续内存空闲块链接在一起的双向链表指针
};
//状态
#define PG_reserved                 0       //表示是否被保留
#define PG_property                 1       //表示是否是空闲块第一页
```

同文件定义了le2page

```
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)
```

将一个链表条目（`list_entry_t`类型）转换回它所属的结构体（struct Page`）的指针，实际上就是将列表转换为页

default_alloc_pages

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) { //检查自由页面和所需页面
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;//遍历
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {   //找到合适的页面，将page指针指向该页面
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link)); //删除链表中的空闲块
        if (page->property > n) {  //拆分
            struct Page *p = page + n; //更新空闲块的起始位置
            p->property = page->property - n;  //更新空闲块的大小
            SetPageProperty(p); //标记空闲块
            list_add(prev, &(p->page_link));  //将新空闲块插入到链表中原始块的前一个位置（局部性）
        }
        nr_free -= n;
        ClearPageProperty(page);  //已分配
    }
    return page;
}
```

default_alloc_pages从空闲内存块中找到第一个比待分配n大的块，如果分配完有剩余，就将剩余的部分插入到链表中，从而实现内存分配。

default_free-pages //释放

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));//没有被保留且没有设置属性，可释
放
        p->flags = 0//清除;
        set_page_ref(p, 0); //引用计数设置为0
    }
    base->property = n;
    SetPageProperty(base);//SetPageProperty宏标记这个页面块为空闲
    nr_free += n;

    //添加到空闲列表
    if (list_empty(&free_list)) {  //头部
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {  //找到位置
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }

    //向前合并
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) { //头节点
        p = le2page(le, page_link);
        if (p + p->property == base) {  //判断相邻
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }
 //向后合并
    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
```

```
    }
}
```

assert语句断言这个基地址所在的页是否为预留，如果不是预留页，那么说明它已经是free状态，无法再次free。之后，遍历空闲链表，如果发现空闲链表当前节点对应的空闲block和待释放block在物理地址上是连续的，则先将空闲链表中对应的空闲block删除，然后合并到待释放block中。遍历完毕后，将待释放block出入到空闲链表的开头即可。

练习2

设计实现过程

该算法的内存初始化和释放和first算法类似，做的改进主要在内存分配过程中，first算法是从头开始遍历空闲链表，找到第一个满足大小的页框块，而best算法则是遍历完整个链表，找到满足条件的最小的块，如果有剩余，就将剩下的加入空闲链表。

改进空间

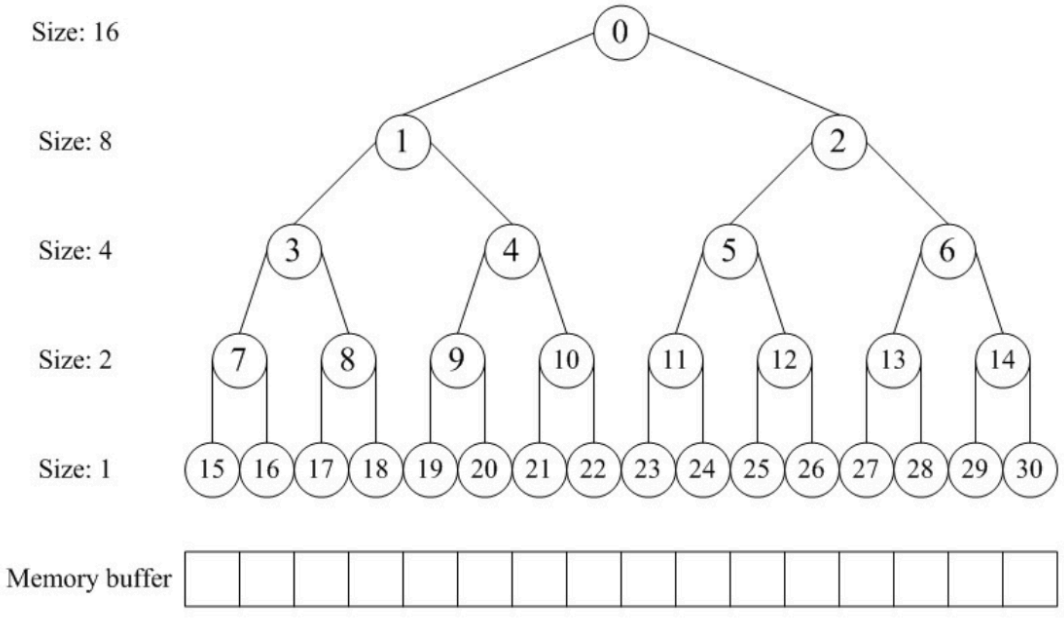
需要遍历整个空闲链表来找到最佳匹配，这在链表很长时可能导致性能下降，时间复杂度为 O(n)，同时，这种算法会产生更多较小的内存碎片，且不能继续利用。

可以考虑用二叉树来存储空闲块，使得查找和插入删除内存块的时间复杂度都降为O(logn)，同时可以将较小的分散内存碎片设法合并为一个大的内存，增加内存的利用率。

challenge1：buddy system（伙伴系统）分配算法

buddy system（伙伴系统）算法原理

通过一个数组形式的完全二叉树来监控管理内存，二叉树的节点用于标记相应内存块的使用状态，高层节点对应大的块，低层节点对应小的块，在分配和释放中我们就通过这些节点的标记属性来进行块的分离合并。如图所示，假设总大小为16单位的内存，我们就建立一个深度为5的满二叉树，根节点从数组下标[0]开始，监控大小16的块；它的左右孩子节点下标[1~2]，监控大小8的块；第三层节点下标[3~6]监控大小4的块……依此类推。



buddy system（伙伴系统）算法过程

内存初始化 buddy_init_memmap

1、初始化从base开始的n个页面，使用 `assert(PageReserved(p))` 确保页面 `p` 已经被预留，将页面的 `flags` 和 `property` 属性清零，调用 `set_page_ref(p, 0)` 将页面的引用计数设为0，调用 `SetPageProperty(p)` 设置页面的属性标志。

```
struct Page *p = base;
    for (; p != base + n; p++)
    {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
        SetPageProperty(p);
    }
```

2、将基页面 `base` 的 `property` 属性设置为 `n`，表示该基页面管理的总页面数，调用 `SetPageProperty(base)` 设置基页面的属性标志，增加全局变量 `nr_free`，表示系统中可用的页面数增加了 `n` 个，将基页面指针 `base` 保存到全局变量 `base0` 中，以便后续使用。计算 `buddy` 数组长度，计算 `buddy` 数组起始地址，初始化 `buddy` 数组。

```
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    //保存起始页面指针
    base0 = base;

    int i = get_power(n);
    //计算buddy数组长度
    length = 2 * (1 << (i));

    unsigned node_size = length;

    //计算buddy数组起始地址
    buddy = (unsigned *)(base + length);

    for (i = 0; i < length; ++i)
    {
        if (is_power_of_2(i + 1))
            node_size /= 2;
        buddy[i] = node_size;
    }
```

分配内存 buddy_alloc_pages

1、利用完全二叉树，二叉树每个节点代表一块内存。

2、`node_size` 存储当前节点代表的页面数量，`length` 是整个内存区域的总页面数，buddy数组代表该节点存储的页面数。

3、分配时从根节点开始向下寻找合适的页面，如果根节点的大小小于请求的页面数 `n`，则设置 `offect` 为-1，这表示没有足够的连续空间来满足请求；向下寻找直到找到一个大小正好等于 `n` 的节点，搜索过程中，优先选择左子节点。

```
//从根节点往下找，直到大小等于n
    for (node_size = length / 2; node_size != n; node_size /= 2)
    {
        //如果左子节点大小大于等于n，移动到左节点
        if (buddy[2 * index + 1] >= n)
        {
            index = 2 * index + 1;
        }
        //反之移动到右节点
        else
        {
            index = 2 * index + 2;
        }
    }
```

4、找到合适的节点后，将其在 `buddy` 数组中的值设为0，表示该节点已被分配。计算分配页面的偏移量 `offect`，用于确定分配页面的具体位置。

```
 //找到的节点标记为已分配
    buddy[index] = 0;

 //计算分配页面的偏移量
 offect = (index + 1) * node_size - length / 2;
```

5、分配完成后，需要从该子节点向上更新所有父节点的大小，将父节点大小更新为其左右节点的最大值。

```
while (index > 0)
    {
        if (index % 2 == 0)
        {
            index = (index - 2) / 2;
        }
        else
        {
            index = (index - 1) / 2;
        }

        //更新父节点大小为较大子节点大小
        buddy[index] = (buddy[2 * index + 1] > buddy[2 * index + 2]) ? buddy[2 *
index + 1] : buddy[2 * index + 2];
     }
```

6、返回指向分配页面的指针page。

释放内存 buddy_free_pages

1、遍历从 `base` 开始的 `size` 个页面，对每个页面 `p`，使用 `assert(!PageReserved(p) && !PageProperty(p))` 确保页面 `p` 没有被预留且没有属性标志，调用 `set_page_ref(p, 0)` 将页面的引用计数设为0，增加 `nr_free`。

```
struct Page *p = base;
    for (; p != base + size; p++)
    {
        assert(!PageReserved(p) && !PageProperty(p));
        set_page_ref(p, 0);
    }
    nr_free += size;
```

2、自底向上查找合适节点，从最小的节点开始，逐步向上查找，直到找到一个大小等于 `n` 的节点。

```
//自底向上查找，直到找到大小等于n的节点
    while (node_size < n)
    {
        node_size *= 2;
        if (index % 2 == 0)
        {
            index = (index - 2) / 2;
        }
        else
        {
            index = (index - 1) / 2;
        }
        if (index == 0)
            return;
    }
buddy[index] = node_size;
```

3、从当前节点向上更新所有父节点的大小，如果 `left + right` 等于 `node_size`，表示两个子节点都可以合并，将父节点的值设为 `node_size`，否则，将父节点的值设为较大的子节点的值。

```
//从当前节点向上更新父节点，直到根节点
    while (index)
    {
        if (index % 2 == 0)
        {
            index = (index - 2) / 2;
        }
        else
        {
            index = (index - 1) / 2;
        }
        node_size *= 2;
        unsigned left = buddy[2 * index + 1];
        unsigned right = buddy[2 * index + 2];
        if (left + right == node_size)
            buddy[index] = node_size;
        else
            buddy[index] = (left > right) ? left : right;
    }
```

测试样例

```
static void
buddy_check(void)
{
    cprintf("buddy check%s\n", "!");
    struct Page *p0, *p1, *p2, *A, *B, *C, *D;
    p0 = p1 = p2 = A = B = C = D = NULL;

    //分别调用 alloc_page() 函数分配三个单个页面 p0、A 和 B
    assert((p0 = alloc_page()) != NULL);
    assert((A = alloc_page()) != NULL);
    assert((B = alloc_page()) != NULL);

    //检查 p0、A 和 B 是否是不同的页面，检查每个页面的引用计数是否为0
    assert(p0 != A && p0 != B && A != B);
    assert(page_ref(p0) == 0 && page_ref(A) == 0 && page_ref(B) == 0);

    //分别调用 free_page() 函数释放 p0、A 和 B 页面
    free_page(p0);
    free_page(A);
    free_page(B);

    p0 = alloc_pages(100);//p0分配100个页面,以下同理
    p1 = alloc_pages(100);
    A = alloc_pages(64);

    // 检验p1和p2是否相邻，并且分配内存是否是大于分配内存的2的幂次
    assert(p1 = p0 + 128);
    // 检验A和p1是否相邻
    assert(A == p1 + 128);

    // 检验p0释放后分配D是否使用了D的空间
    free_page(p0);
    D = alloc_pages(32);
    assert(D == p0);

    // 检验释放后内存的合并是否正确
    free_page(D);
    free_page(p1);
    p2 = alloc_pages(256);
    assert(p0 == p2);

    free_page(p2);
    free_page(A);
}
```

测试结果

```
he@he-virtual-machine:~/NKU_OS/riscv64-ucore-labcodes/lab2$ make qemu

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    ____  _____ _____
  / __ \                  / ___|| ___ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200032 (virtual)
  etext  0xffffffffc0201a3c (virtual)
  edata  0xffffffffc0206030 (virtual)
  end    0xffffffffc02064b8 (virtual)
Kernel executable memory footprint: 26KB
memory management: buddy_system_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
buddy check!
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
```

测试正确，buddy system伙伴系统算法成功实现。

challenge2：slub分配算法

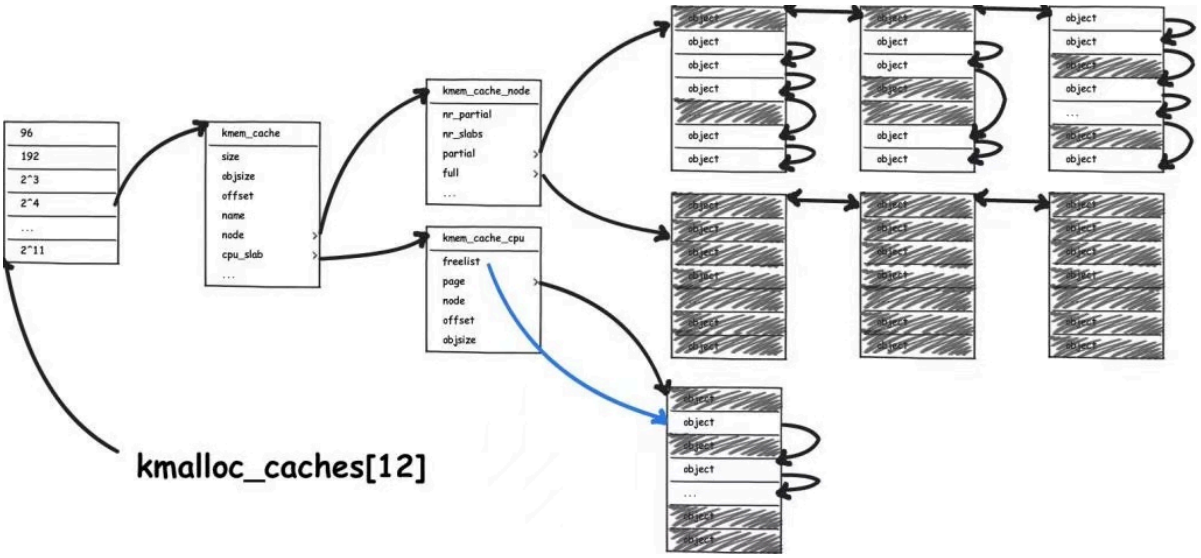
任意大小的内存单元slub分配算法（需要编程）

算法介绍

SLUB是一种用于管理内核内存块分配的分配器，在内核中用于高效地分配和释放大小可变的小内存块，主要用于内核对象的分配。

在分配前，先判断需求的块大小，对于较大的块直接使用原本的页面分配算法；而对于小块，则取出一页中的一小块来使用。

以下内容参考博客linux 内核 内存管理 slub算法 （一） 原理-CSDN博客，对slub算法原理进行描述。



上图是slub算法的一个基本描述，从`kmalloc_caches`开始，该数组的每个元素是一种特定大小的页面。对于每一种页面，都存在一个`kmem_cache`，像是一个管理员，负责把该大小的页面“零售”出去。

我们把每一个大页称为一个`slab`，一个`slab`中存在多个`object`供分配。

`kmem_cache`中维护了两个结构体，`kmem_cache_node`与`kmem_cache_cpu`。`kmem_cache_node`是一个“仓库”，他主要维护了两个链表，`partial`链表中维护了那些部分被使用的`slab`，`full`链表中维护了多个已全部被分配的`slab`。`kmem_cache_cpu`是一个“前台”，它维护一个当前正在被“售卖”的`slab`，其中`freelist`指向下一个空闲的obj块。当申请到来时，`kmem_cache_cpu`将空闲的`object`分配出去，如果当前`slab`已满，就从仓库中换出一个半空的`slab`供使用。

细节描述

- 申请到来，查看需要大小。大内存直接调用分页算法，小内存时用slub。
- 第一次申请，`kmem_cache_node`与`kmem_cache_cpu`都为空。申请出一个全新的slab，将其中一个`object`分配给用户，剩余的链入`kmem_cache_cpu`的`page`。更新`freelist`指向下一个空闲的obj块。
- 再申请：
  - `kmem_cache_cpu`上有空闲obj。直接分配。
  - `kmem_cache_cpu`上无空闲obj，但`kmem_cache_node`的`partial`中有。将`kmem_cache_cpu`当前的`slab`放入`full`中，并从`partial`中取出一个`slab`进行使用。将其中一个`object`分配给用户，剩余的链入`kmem_cache_cpu`的`page`。更新`freelist`指向下一个空闲的obj块。
  - `kmem_cache_cpu`与`kmem_cache_node`都无空闲obj。与第一次申请的情况类似，申请出一个全新的slab，将其中一个`object`分配给用户，剩余的链入`kmem_cache_cpu`的`page`。更新`freelist`指向下一个空闲的obj块。
- 释放：
  - `kmem_cache_cpu`当前维护的`slab`即为要释放的obj所在。直接链入，放回。
  - 不是所在，要将obj放回其所在的、`kmem_cache_node`中的`slab`：
    - `slab`为`full`。放回后，将当前`slab`从`full`中卸下链入`partial`中。
    - `slab`为`partial`。直接链入，放回。

代码实现

由于这个算法比较复杂，本次实验中只做了一个非常简化的基本实现，参考了**kmalloc**算法，简单的将分配分为两层，分别为页与小块维护一个链表。

原本的slub算法中有12种不同大小的slab，本次实验中只用一页表示，一页的**PGSIZE**在mmu.h中被定义为4096字节。

对于小块的分配，去除了`kmem_cache_cpu`与`kmem_cache_node`等的分配管理，只用多个**object**简单的连接为一个循环链表。

```
// object小块
struct object {
    int objsize;  // 供分配的块大小，objsize个OBJ_UNIT的大小
    struct object *next;  // 下一个块
};
typedef struct object obj_t;

#define OBJ_UNIT sizeof(obj_t)  // 小块的总大小 16字节
#define OBJ_UNITS(size) (((size) + OBJ_UNIT - 1)/OBJ_UNIT)  // 转换size为需要的块数，向上取整

// bigblock大块
struct bigblock {
    int order;  // 阶数，说明大小
    void *pages;  // 这个页的起始地址
    struct bigblock *next;  // 下个页
    bool is_bigblock;  // 标识是否为大块
};
typedef struct bigblock bigblock_t;

static obj_t arena = { .next = &arena, .objsize = 1 };  // 小块链表头，初始时指向自身。
static obj_t *objfree = &arena;  // 当前可用的空闲小块链表，初始指向arena。
static bigblock_t *bigblocks_head = NULL;  // 大页的链表头。
```

分配算法

比较重要的是小块的分配算法：

- 大小符合要求，遍历小块链表，找到一个大于等于当前所需size的obj。
    - 如果大小相等，就直接取出。
    - 如果比需要的大，就取下需要的部分，将剩余部分放回链表。
- 如果遍历结束仍未找到符合要求的obj，说明需要一个新的slab了，使用`alloc_pages(1)`来申请一页新内存。这个函数是前面实验编写的。新内存会被链入链表，cur指向它，再开始一轮循环，这一次就能找到足够大的obj了。

```
// 小块分配(传入size是算上头部obj_t的)
static void *obj_alloc(size_t size)
{
    assert(size < PGSIZE);

    obj_t *prev, *cur;
    int objsize = OBJ_UNITS(size);
```

```
    prev = objfree;  // 从头遍历小块链表
    for (cur = prev->next; ; prev = cur, cur = cur->next) {

        if (cur->objsize >= objsize) { // 如果当前足够大
            // 刚刚好，直接从链表中取出
            if (cur->objsize == objsize)
                prev->next = cur->next;
            // 比需要的大，取下需要的部分，剩下的继续放入链表
            else {
                // 一个指针加法，在当前块指针的基础上偏移units个SLOB单位
                // 使得prev->next指向剩余块的起始位置，将当前块分为两部分
                prev->next = cur + objsize;

                prev->next->objsize = cur->objsize - objsize;
                prev->next->next = cur->next;  // 剩余块被连入链表
                cur->objsize = objsize;  // units大小的块被取出
            }
            objfree = prev;  // 更新当前可用的空闲小块链表位置
            return cur;
        }

        if (cur == objfree) {  // 链表遍历结束了，还没找到。需要扩展内存

            if (size == PGSIZE){return 0;} // 应该直接分配一页，不予扩展

            // call pmm->alloc_pages to allocate a continuous n*PAGESIZE
            cur = (obj_t *)alloc_pages(1);
            if (!cur)  // 如果获取失败，返回 0
                return 0;

            obj_free(cur, PGSIZE);  // 将新分配的页加入到空闲链表中
            cur = objfree;  // 从新分配的页再次循环
        }
    }
}
```

完整的slub分配流程：

- 传入申请的size，若size+slot_t小于一页大小，调用`obj_alloc(size_t size)`进行小块分配。
- 若大于等于一页，则先调用`obj_alloc(size_t size)`为`bigblock_t`分配一块小内存，该内存的起始位置就是这个大块的位置，然后调用页分配算法`alloc_pages(order)`，为其分配阶数个连续的内存页，然后插入大块链表的头部`bigblocks_head`。

```
// 总slub分配算法(传入申请的大小)
void *slub_alloc(size_t size)
{
    obj_t *m;
    bigblock_t *bb;

    // 申请size+slot_t小于一页，用小块分配。
    // 如果分配成功，则返回 (void *)(m + 1)，跳过了一个 obj_t 的单位,指向实际可用的内存地
址。
    // 如果分配失败，返回 0
```

```
    if (size < PGSIZE - OBJ_UNIT) {
        m = obj_alloc(size + OBJ_UNIT);
        return m ? (void *)(m + 1) : 0;
    }

    // 为bigblock_t结构体先分配一小块
    bb = obj_alloc(sizeof(bigblock_t));
    if (!bb)
        return 0;

    // 分配大页
    bb->order = ((size - 1) >> PGSHIFT) + 1;  // PGSHIFT为12，向右移12位的效果相当于除以2^12即4096）
    bb->pages = (void *)alloc_pages(bb->order);  // 分配2^order个页

    // 设置大块标志
    bb->is_bigblock = 1;

    // 分配成功，将其插入到大块链表的头部。
    if (bb->pages) {
        bb->next = bigblocks_head;
        bigblocks_head = bb;
        return bb->pages;
    }

    // 如果页分配失败，释放之前为bigblock_t结构体分配的内存。返回0。
    obj_free(bb, sizeof(bigblock_t));
    return 0;
}
```

释放算法

比较重要的是小块的释放算法：

- 遍历当前的obj链，按地址找到合适的位置插入要释放的`object`。
- 检查该`object`前后相邻位置是否有空闲块，若有则合并。否则就直接在该位置插入链表。

```
static void obj_free(void *block, int size)
{
    obj_t *cur, *b = (obj_t *)block;

    if (!block)
        return;

    if (size)
        b->objsize = OBJ_UNITS(size);

    // 从objfree开始遍历，寻找到合适的cur位置插入，b要在cur和cur->next之间
    for (cur = objfree; !(b > cur && b < cur->next); cur = cur->next)
        // b正好在环状列表开头末尾的特殊情况
        if (cur >= cur->next && (b > cur || b < cur->next))
            break;
    // 如果b和后面的相邻，与后面合并
    if (b + b->objsize == cur->next) {
```

```
        b->objsize += cur->next->objsize;
        b->next = cur->next->next;
    } else  // 否则b的右侧指针连入链表
        b->next = cur->next;
    // 如果b和前面相邻，与前面合并
    if (cur + cur->objsize == b) {
        cur->objsize += b->objsize;
        cur->next = b->next;
    } else  // 否则curnext指向b（b前侧连入链表）
        cur->next = b;

    // 更新空闲块位置
    objfree = cur;
}
```

完整的slub释放流程：

- 判断当前要释放的块是否为大块（判断内存地址是否按页大小对齐，并遍历链表再次检查），如果为大块，就从大块链表中移除当前节点，并调用之前实验中的页释放算法`free_pages()`，还要调用小块释放算法释放掉块头`bigblock_t`结构体。
- 如果没有进入大块释放的流程，说明是一个小块释放，调用`obj_free`进行释放。

```
// 总slub释放算法
void slub_free(void *block)
{
    // bb用于遍历记录大块内存的链表。
    // last用于指向链表中前一个节点的next指针
    bigblock_t *bb, **last = &bigblocks_head;

    if (!block)
        return;

    // 判断是否是大块
    // 遍历大块链表
    for (bb = bigblocks_head; bb != NULL; last = &bb->next, bb = bb->next) {
        if (bb->pages == block && bb->is_bigblock) {
            // 确认是大块
            *last = bb->next;  // 从链表中移除当前节点
            // call pmm->free_pages to free a continuous n*PAGESIZE memory
            free_pages((struct Page *)block, bb->order);  // 释放大块页
            obj_free(bb, sizeof(bigblock_t));  // 释放 bigblock_t 结构体
            return;
        }
    }

    obj_free((obj_t *)block - 1, 0);
    return;
}
```

测试样例

编写一段检查代码：

```
void slub_check()
{
    cprintf("slub check begin\n");
    print_objs();


    cprintf("alloc test start:\n");
    // 测试申请
    cprintf("p1 alloc 4096\n");
    void* p1 = slub_alloc(4096);
    print_objs();  // 输出object链表信息的函数

    cprintf("p2 alloc 2\n");
    void* p2 = slub_alloc(2);
    print_objs();

    cprintf("p3 alloc 32\n");
    void* p3 = slub_alloc(32);
    print_objs();


    cprintf("free test start:\n");
    // 测试释放
    cprintf("free p1\n");
    slub_free(p1);
    print_objs();

    cprintf("free p3\n");
    slub_free(p3);
    print_objs();

    cprintf("slub check end\n");
}
```

运行结果如下：

```
slub_init() succeeded!
slub check begin
objsizes: Total number of objs: 0

alloc test start:
p1 alloc 4096
objsizes: 254 Total number of objs: 1

p2 alloc 2
objsizes: 252 Total number of objs: 1

p3 alloc 32
objsizes: 249 Total number of objs: 1

free test start:
free p1
objsizes: 2 249 Total number of objs: 2

free p3
objsizes: 2 252 Total number of objs: 2

slub check end
```

初始状态下，obj数为0，objsizes无信息输出。

申请测试：

申请一个4096字节大小的p1：

- 是第一次申请，因此obj链表中多了一个块，大小为256：OBJ_UNIT大小为16字节，256*16=4096，验证了申请一页。
- 输出254的原因是：申请4096字节会分配一个大块，大块申请会用掉一个`bigblock_t`的空间（32字节），即从该obj中取出**两个OBJ_UNIT**，该obj剩余可用大小254，结果正确。

申请一个2字节大小的p2：

- 刚刚申请的obj空间足够，会用掉2个`OBJ_UNIT`的空间（2+16=18，向上取整为2个`OBJ_UNIT`），剩余大小为可用252，而obj数量不变，结果正确。

申请一个32字节大小的p2：

- 刚刚申请的obj空间足够，会用掉3个`OBJ_UNIT`的空间（32+16=48，取整为3个`OBJ_UNIT`），剩余大小为可用249，而obj数量不变，结果正确。

释放测试：

释放掉p1：

- 由于p1与空闲obj之间间隔着p2与p3，所以不会被合并，而是作为一个单独的`object`被链入链表，所以obj数量变为2，大小分别为2（p1大小）、249。结果正确。

释放掉p3：

- 由于p3与剩余的249大小的空闲obj相邻，所以会被合并，obj数量仍为2，大小分别为2、252（249+3）。结果正确。

challenge3

如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围?

opensbi在进入内核前将扁平化的设备树地址写入内存地址，设备树源文件dts内容中通过节点将设备内存在的硬件存储，每个节点通常带有compatible属性字符串用来在驱动初始化时枚举设备探测匹配使用，如果驱动和设备树中的节点完成了匹配，则会根据节点信息中的内容初始化驱动，被bootloader传入到内核中。随后 OpenSBI 会将其地址保存在 a1 寄存器中，给我们使用,得到物理内存所在的物理地址。