Lab2 实验报告：物理内存管理

2310364 柳昕彤 2313310 熊诚义 2311887 陈语童

练习1：理解 first-fit 连续物理内存分配算法

一、对于原代码按顺序逐行梳理，并分析每个函数作用

1. 主体代码前

`#include` 一系列库文件和头文件。

```
static free_area_t free_area;
#define ......
```

▲ 观察 `memlayout.h` 中结构体 `free_area_t` 的定义，了解到该结构体用来维护一个记录单个 pmm 管理器中未使用的 page 的 **双向链表** 结构（在实验手册中已经详细介绍双向链表的实现，准确来说是“环状双向链表”）。下面两个宏定义分别对应结构体中定义到的两个成员变量，分别是双向链表本身和空闲页的总数，宏定义使得下面的访问更加便利。

2. default_init 函数

```
list_init(&free_list);
nr_free = 0;
```

▲ 封装地调用 `list_init` 初始化空闲链表，并将空闲页数置零。

3. default_init_memmap 函数

这个函数是在根据传入的内存范围（起始页和页数）建立空闲页链表。

首先断言监测输入有效性。

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

▲ 然后在循环中，依次——先检查页是否是被系统保留（Reserved）为可用状态的，然后清空引用数（ref，表示这个页正在被多少个地方访问）。

```
base->property = n;
SetPageProperty(base);
nr_free += n;
```

▲ 在接下来标记块头页（base）为空闲/可分配（property 相关的意义在指导书中已经说明，property 参量为以当前页为头的块内有多少个连续的页）。然后空闲页数加上这次被初始化的页数量。

```
if (list_empty(&free_list)) {
    list_add(&free_list, &(base->page_link));
} else {
    list_entry_t* le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        ......
    }
}
```

▲ 接下来一块的 `if-else` 分支语句用于将这次初始化的空闲页放入空闲双向链表中，主要逻辑是：(1) 如果双向列表现在为空，直接插入即可；(2) 如果不空，遵循地址顺序，从头遍历双向链表，直到找到正确的地址，插入。注意这里遍历时候使用到了指导书中提到的，特别设计的**抽象**的双向链表节点 `list_entry`，可以适配到任何实际的数据类型/结构。

4. default_alloc_pages 函数

★ 这个函数是整个算法的核心逻辑实现，按照程序给出的要求从链表中分配出合适的连续 **物理页内存**，给程序使用。

首先仍然进行参数检查，以及如果可用页不足，直接返回 `NULL`.

接下来是 **“first-fit 连续物理内存分配算法”** 的核心逻辑实现：

```
struct Page *page = NULL;
list_entry_t *le = &free_list;
while ((le = list_next(le)) != &free_list) {
    struct Page *p = le2page(le, page_link);
    if (p->property >= n) {
        page = p;
        break;
    }
}
```

▲ 使用了 `list_entry` 进行双向空闲列表的遍历，并且在遍历过程中不断进行判断，找到 **第一块能够满足大小要求的空闲块**，也就是一块页内存（就是 “first-fit” 的实际含义）。

接下来对于找到合适块的情况，进行从链表中取出块的相关处理。首先将块从链表中取下（代码略）。

```
if (page->property > n) {
    struct Page *p = page + n;
    p->property = page->property - n;
    SetPageProperty(p);
    list_add(prev, &(p->page_link));
}
```

▲ 当代码块大小超过要求的大小时，拆分块，将超过要求的部分重新作为一个新的块插入到链表中，使用到了 `list_add`.

而后进行分配后一些全局状态的修改，按需减少了总空闲页数，并清空了分配出去的页的 property 标志，表示它不再空闲可用于分配。最后返回分配出来的页，注意返回的是这一块连续内存的**首地址**。

5. default_free_pages 函数

用于释放一定数量的目前不再使用的连续物理页内存资源。

首先依旧检查传参正确，以及要清零的页确实都是已经分配过的。然后预先清零标志和 ref 引用数量。

```
base->property = n;
SetPageProperty(base);
nr_free += n;
```

▲ 这里继续去除要清除页的被分配状态，分别把当前连续内存首个页标记为空闲块、更新空闲页的总数。

下一块 `if-else` 分支语句是把新的空闲块插入双向链表，实现原理和上一函数 `default_alloc_pages` 中的插入是一样的。

下面分别进行插入后的前后合并尝试。
前合并：

```
list_entry_t* le = list_prev(&(base->page_link));
if (le != &free_list) {
    p = le2page(le, page_link);
    if (p + p->property == base) {
        p->property += base->property;
        ClearPageProperty(base);
        list_del(&(base->page_link));
        base = p;
    }
}
```

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));
    }
}
```

核心的逻辑都是：

- 先取前一个或后一个 `list_entry` 元素，如果不是表头就可能可以发生合并；
- 接着判断两个待合并的两个块中物理地址较前的一个，加上 property（也就是有多少个连续的页）后，是否是下一块的首地址，这一结果成立就说明两个块是在物理内存意义上紧接着连续的，那么就可以发生合并；
- 然后将前一个块的 property 加上后一个块的 property，再调用 `list_del` 删掉后一个块，就得到了新的、合并后的、更大的块。

6. default_nr_free_pages 函数

只是一个简单的封装函数，返回当前空闲页的个数。

7. 在这之后

是对于 **“first-fit 连续物理内存分配算法”** 的测试函数。

- `basic_check` 测试最基础的页分配、释放，以及 `free_list` 的功能是否正常。
- `default_check` 测试更综合更复杂，如大量页的分配 -> 块拆分、释放 -> 块合并等，拥有一个更加完整的页管理流程。

最后还有 `pmm_manager` 的实例化，手册中介绍比较详细了不再赘述。

二、一些细节

1. PageReserved -> 一系列同类宏定义函数

在 `memlayout.h` 中定义了一些列函数：

```
#define SetPageReserved(page)       ((page)->flags |= (1UL << PG_reserved))
#define ClearPageReserved(page)     ((page)->flags &= ~(1UL << PG_reserved))
#define PageReserved(page)          (((page)->flags >> PG_reserved) & 1)
#define SetPageProperty(page)       ((page)->flags |= (1UL << PG_property))
#define ClearPageProperty(page)     ((page)->flags &= ~(1UL << PG_property))
#define PageProperty(page)          (((page)->flags >> PG_property) & 1)
```

▲ 其中包含了上面代码实现中的 `PageReserved` 函数。
而在此之前，对 reserved 标志位和 property 标志位在 flag 中的位置进行了划分，分别在低位起的第0和第1位：

```
#define PG_reserved                 0       // if this bit=1: the Page is reserved for kernel, cannot be used in alloc/free_pages; otherwise, this bit=0
#define PG_property                 1       // if this bit=1: the Page is the head page of a free memory block(contains some continuous_addrress pages), and can be used in alloc_pages; if this bit=0: if the Page is the the head page of a free memory block, then this Page and the memory block is alloced. Or this Page isn't the head page.
```

▲ 注释也说明了，`reserved = 0` 说明页处于可分配给程序使用的状态，否则不能够分配；`property = 1` 说明当前页是当前物理内存块的头页，否则有两种情况：(1) 头页代表的物理内存块已被分配；(2) 当前页不是头页。

那么上面那些宏函数的实现是如何的?
它们以定义的标志位位置为左移/右移位数，然后与页本身的 flag 或单 bit 立即数进行位运算，便可以对页 flag 进行更新或监测当前 flag 中对应的标志位是0还是1.

```
1UL是什么？
U = Unsigned：无符号数
L = Long：长整型
——无符号长整型1
不直接写作1（默认int型），是为了防止位移导致溢出/未定义行为。
```

比如：`#define PageReserved(page)          (((page)->flags >> PG_reserved) & 1)`
这时 flag 左移0位，`&` 只会将最低位进行与运算，那么就可以运算监测 flag 的最低位，也就是 reserved 标志位是0还是1.

2. set_page_ref -> 什么是 ref?

ref 是 Page 结构体中的一个成员变量，当 `ref > 0` 时，页被 “引用/正在使用”，不能够回收释放或当作空闲页；当 `ref == 0` 时，没有任何地方正在使用，那么就可以释放作为空闲页（其它标志也允许的前提下）。

在 `pmm.h` 中可以看到，除了 `set_page_ref` 函数之外，还有配套的 `page_ref_inc` 和 `page_ref_dec` 分别用来增加和减少页面的被引用次数，这在实际多进程运行中是重要的。

常见的 “引用” 来源有（询问了大模型帮助了解）：

- 页表中有虚拟页映射到该物理页，一个映射对应一个引用。
- 内核数据结构（缓存、内存池等）持有该页。
- DMA/外部设备正在访问页面。
- 作为页面缓存的一部分被多个用户共享。

3. le2page 转化

在上面的代码实现中，使用到了 le2page 宏定义，能够将 `list_entry` 向上一级转化到对应的 Page 结构体。实验指导书中已经有比较详细的介绍

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

```
#define to_struct(ptr, type, member)                              \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

```
#define offsetof(type, member)                                       \
    ((size_t)(&((type *)0)->member))
```

▲ 使用层级递进的宏定义：先假设地把结构体放在0地址，得到 `list_entry` 的 offset，而后将 `list_entry` 指针减去 offset 得到结构体起始地址，就可以最终将 `list_entry` 转化成 Page 结构体。

4. list_init 在做什么?

很简单，创建一个新的环状双向链表记录空闲页。

```
static inline void
list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}
```

实际上，环状双向链表需要一个不对应实际页内存块的节点用来表示表头（也表示表尾，因为是环状）。在遍历开始时，以及回收页面时判断能否合并等工作时都需要利用其作为切入点或进行边界判断。在上面新建链表的代码中，就是先仅仅创建出了这个并没有有效数据的节点，此时链表为空。

```
static inline bool
list_empty(list_entry_t *list) {
    return list->next == list;
}
```

▲ 判断链表为空的逻辑也可以证明上面的设计推理：为空的条件是列表起始 `list_entry`（全部用 `list` 表示）的下一个 `list_entry` 还是它本身。上面初始化中 `elm->prev = elm->next = elm` 的赋值导致这个条件一定被满足，说明上面刚创建出来的就是没有数据的链表，为空。

三、整体工作流程归纳

1. 内核启动到整个物理内存初始化结束，函数/结构体的调用过程：

`kern_init --> pmm_init --> page_init --> init_memmap --> pmm_manager --> init_memmap`
通过 page_init 解析可以自由使用的物理内存区间，通过 init_memmap 初始化这一整块自由物理内存区间，pmm 用来查询当前使用内存分配策略下函数的实际地址。
在 first-in 的方法下，pmm_manager 被实例化为 default_pmm_manager，init_memmap 也被相应地赋值（指针的赋值）到 default_init_memmap：

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

2. 初始化之后的具体操作

初始化完成之后就可以由程序申请页获得运行空间，这中间 `alloc_pages` 和 `free_pages` 将被频繁地交叉调用，按照 first-fit 的方式分配页，并释放回收不再使用的页。当然这里的两个函数也已经在实例化过程中被函数指针地赋值为 `default_···` 方法。此外还会发生 `nr_free_pages` 的查询。

四、原代码改进空间

1. 代码复用改进

在 `default_init_memmap` 和 `default_free_pages` 出现了完全相同的两段 “插入新的空闲页到双向链表中” 的逻辑。这一块代码可以进一步 **封装为函数**，避免代码复用而造成冗余。

2. 缺少锁机制

在实际的操作系统运行环境下，有多个进程来申请页面，将会导致冲突。这时候需要给分配页、向链表插入页块和从链表卸下页块等原子性操作加锁，保证线程安全。包括 ref 的更新等，也可使用 atomic 库的方法来做。

3. 算法设计与策略“本质上”的改进

first-fit 本质上并不是一个效率高的算法，因为它在最坏的情况下每次都需要遍历全部链表元素，时间复杂度是 $O(n)$ 级别的。可以改进算法，使用更快的 best-fit 或者 next-fit 等，优化链表的排序方法、分配块的查找策略。

4. 算法设计与策略“非本质上”的改进

和上一条同样的出发点，可以通过一些额外的数据结构来优化链表访问。比如升级为维护一个双向的 **跳表**，虽然会带来额外的空间代价，并且会在插入和卸下的时候消耗更多时间，但相对于 $O(n)$ 到 $O(log\ n)$ 级别时间复杂度的优化，访问效率还是提升了。

5. 宏定义可能带来的不便

```
static free_area_t free_area;
#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)
```

对于这样的宏定义，会直接暴露 `free_area` 的内部成员；而且如果对于成员名进行修改，不会有编译器类型检查，容易导致隐藏的 bug 出现。所以从上面两点考虑，更好的方法还是回到封装函数访问。

练习2：实现Best-Fit连续物理内存分配算法

设计实现过程

设计思路

Best-Fit算法的核心思想是：**在所有能满足需求的空闲块中，选择大小最接近需求的那一个。**

与First-Fit的本质区别：

- **First-Fit**：找到第一个满足大小的块就立即停止，时间复杂度最好情况O(1)
- **Best-Fit**：必须遍历整个空闲链表，找到最小的满足块，时间复杂度O(n)

设计目标：

1. 减少大块的浪费，保留大块用于大请求
2. 通过精确匹配减少内存碎片
3. 保持链表按地址有序，便于合并操作

数据结构设计：

- 使用双向链表管理空闲块（与First-Fit相同）
- 每个空闲块的首页用property记录块大小
- 维护全局计数器nr_free记录总空闲页数

实现代码与说明

1. `best_fit_init_memmap()` 实现

```
static void best_fit_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            // 找到第一个大于base的页，将base插入到它前面
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
```

```
            } else if (list_next(le) == &free_list) {
                // 已经到达链表结尾，将base插入到链表尾部
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

实现要点：

- 初始化每个页面的flags、property、引用计数
- 按地址顺序插入链表，保持链表有序
- **注意**：在插入到链表末尾时必须加`break`，否则会继续无意义的循环

2. `best_fit_alloc_pages()` 实现（核心）

这是Best-Fit算法的**核心部分**，体现了"最佳适配"的思想。

```
static struct Page *best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;  // 初始化为一个不可能的大值

    // 遍历整个空闲链表，找到最小的满足需求的块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        // 如果当前块满足需求且比之前找到的块更小
        if (p->property >= n && p->property < min_size) {
            page = p;
            min_size = p->property;
        }
    }

    // 分配找到的最佳块
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));

        // 如果块较大则分割
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }

        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

关键改动与实现细节：

1. **初始化最小值**：`min_size = nr_free + 1`

- 设为不可能达到的大值，确保第一个满足条件的块能被记录

2. **完整遍历策略**：

```
if (p->property >= n && p->property < min_size) {
    page = p;
    min_size = p->property;
}
```

   - **不能**在找到第一个满足的块时就break（这是First-Fit的做法）
   - 必须遍历完整个链表，找到所有满足条件中最小的
   - 每次发现更小的满足块时更新`page`和`min_size`

3. **分割剩余块**：

```
if (page->property > n) {
    struct Page *p = page + n;
    p->property = page->property - n;
    SetPageProperty(p);
    list_add(prev, &(p->page_link));
}
```

   - 如果最佳块大于需求，将剩余部分作为新块插回链表
   - 插入到原块的前驱位置，保持地址有序

3. `best_fit_free_pages()` 实现

```
static void best_fit_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));
        p->flags = 0;
        set_page_ref(p, 0);
    }

    // 设置当前页块的属性
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    // 按地址顺序插入链表
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            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));
```

```
            }
        }
    }

    // 尝试向前合并
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        // 判断前面的空闲页块是否与当前页块连续
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    // 尝试向后合并
    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));
        }
    }
}
```

实现要点：

- 重置页面状态并设置块属性
- 按地址顺序插入链表
- 向前和向后尝试合并相邻的空闲块
- **注意**：合并后要更新base指针，以便继续向后合并

物理内存分配和释放的详细流程

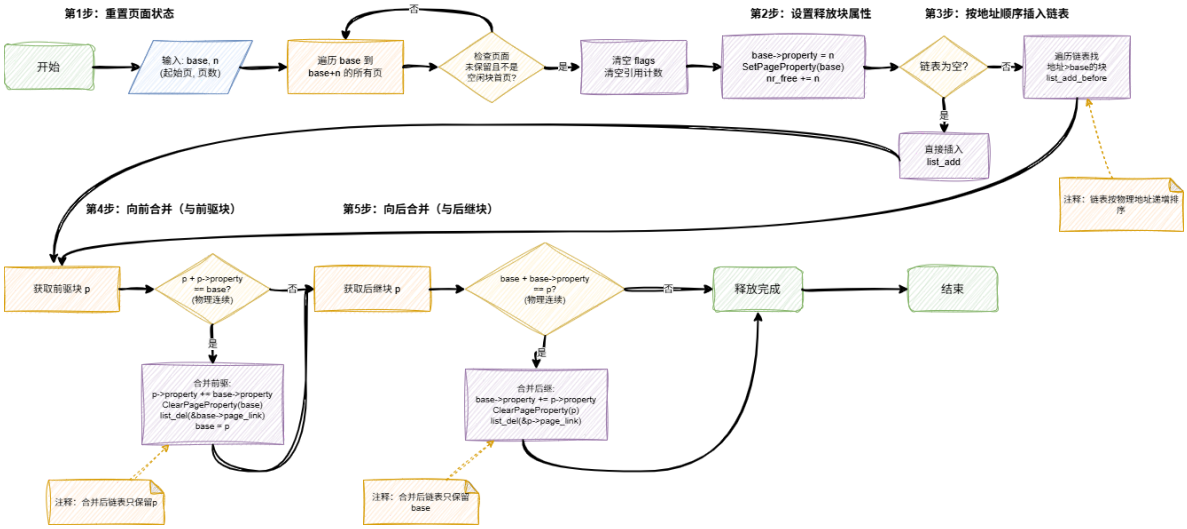
内存分配流程（best_fit_alloc_pages）

 page_allocation_flowchart

关键点说明：

- **Best-Fit的本质**：第2步必须遍历完整个链表，不能提前退出
- **分割策略**：精确分配所需页数，剩余部分返回链表，避免内部碎片
- **地址有序性**：剩余块插入到原块位置，维护链表的地址有序特性

内存释放流程（best_fit_free_pages）



合并操作的关键逻辑：

```
// 判断物理地址连续的条件
p + p->property == base  // 前驱块的末尾 = 当前块的开头

base + base->property == p  // 当前块的末尾 = 后继块的开头
```

合并的优势：

- 减少外部碎片
- 维护更少的空闲块节点
- 提高后续大块分配的成功率

测试与验证

编译并运行测试：

```
make clean
make
make qemu
```

预期输出应包含：

- `check_alloc_page() succeeded!`
- `best_fit_check() succeeded!`

```
make grade
```

```
>>>>>>>>>> here_make>>>>>>>>>>
<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=4820
<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
Total Score: 25/25
```

这表明Best-Fit算法实现正确，通过了所有测试用例。

进一步改进

将空闲链表改为**按块大小从小到大排序**后，分配操作的效率可以显著提升。当前实现采用地址排序，每次分配时必须遍历整个链表来找到最小的满足块，时间复杂度始终为O(n)。而按大小排序后，由于链表已经有序，第一个满足条件（`property >= n`）的块就是最小的满足块，可以立即返回，无需继续遍历。特别是在完美匹配的情况下（请求大小正好等于某个空闲块），性能提升更为明显，可以直接在链表前部快速定位，避免了全链表扫描。

按大小排序使得Best-Fit算法的核心逻辑更加简洁和符合直觉——"找到第一个满足条件的块即为最佳匹配"，消除了当前实现中需要维护`min_size`变量并持续更新的复杂性。同时，这种排序方式还带来了额外的优势：小块集中在链表前部，提高了CPU缓存命中率，因为大多数分配请求会频繁访问链表头部区域。虽然这种改进会增加释放操作的复杂度（合并后可能需要重新调整块在链表中的位置），但考虑到操作系统中分配操作的频率通常远高于释放操作，总体性能提升仍然显著且能更精确地匹配请求大小，从而减少内存碎片。

扩展练习Challenge：buddy system（伙伴系统）分配算法

设计原理和核心思想：

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理,系统中的所有内存块（无论是空闲还是已分配）的大小都是2的幂次方个页帧（比如1, 2, 4, 8, ..., 1024个页）， 所以我们划分的每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

伙伴关系：两个大小相同、地址相邻且满足特定对齐条件的内存块被称为“伙伴”。合并和分裂操作都围绕伙伴关系进行。

设计思路：

我们要编写的分裂过程是当需要分配一个大小为 `n` 的内存块时，系统会查找大小为 `2^k` 的最小空闲块，其中 `2^k >= n`。 如果找到了大小正好为 `2^k` 的块，则直接分配。如果找到的块大于 `2^k`（例如 `2^m`，其中 `m > k`），系统会将其**分裂**成两个大小为 `2^(m-1)` 的伙伴块。一个用于分配或继续分裂，另一个加入到对应大小的空闲链表中。这个过程会递归进行，直到得到我们所需大小的块。

当一个内存块被释放时，我们要检查其伙伴块是否也处于空闲状态。如果伙伴块也空闲，系统会将它们**合并**成一个大小翻倍的父块。这个合并过程会递归进行，直到父块的伙伴不空闲，或者已经合并到最大的块。

> 举个例子：假设进行需要操作系统分配一个1页大小的块，但只有一块16页的空闲块。

```
+-------------------------------------------------------------+
|                        16 pages (Order 4)                   |
+-------------------------------------------------------------+
                       |
                       v (分裂)
+------------------------------+------------------------------+
|       8 pages (Order 3)      |       8 pages (Order 3)      | <- 右边伙伴加入空闲链表
+------------------------------+------------------------------+
        |
        v (分裂)
+--------------+--------------+
| 4 pages (Ord 2)| 4 pages (Ord 2)|      <- 右边伙伴加入空闲链表
```

```
+---------------+---------------+
|
V (分裂)
+-------+-------+
| 2(O1) | 2(O1) |                <-右边伙伴加入空闲链表
+-------+-------+
|
V (分裂)
+---+---+
| 1 | 1 |                    <- 右边伙伴加入空闲链表
+-|-+---+
|
V (分配)
分配这1页
```

当一个内存块被释放时，我们要检查其伙伴块是否也处于空闲状态。如果伙伴块也空闲，系统会将它们**合并**成一个大小翻倍的父块。这个合并过程会递归进行，直到父块的伙伴不空闲，或者已经合并到最大的块。

> 举个例子：假设要释放一个1页大小的块 `A`，其伙伴 `B` 也空闲。

```
+---+---+----------------------+-------------------------------+
| A | B |       (free)         |            (free)             |
+-|-+---+----------------------+-------------------------------+
| (释放A)
V
检查B是否空闲 -> 是
+-------+----------------------+-------------------------------+
| A   B |       (free)         |            (free)             | <- 合并A和B
+-------+----------------------+-------------------------------+
|
V
检查(A&B)的伙伴C是否空闲 -> 是
+---------------+-----------------------------------------------+
|   (ABC)       |                  (free)                       | <- 继续合并
+---------------+-----------------------------------------------+
... 直到伙伴不空闲或达到最大块
```

实验代码

1. 数据结构设计

为了高效地管理不同大小的空闲块，我们使用一个数组，其每个元素是一个双向链表。

```c
\#define MAX_ORDER 11  // 支持最大2^10 = 1024页
typedef struct {

  list_entry_t free_list[MAX_ORDER];  // 每个order一个链表

  unsigned int nr_free[MAX_ORDER];   // 每个order的空闲块数量

} buddy_free_area_t;

static buddy_free_area_t buddy_free_area;
```

`free_list[i]`是一个双向链表的头节点，链接了所有大小为 `2^i` 页的空闲块。

- `Page` 结构体中的 `property` 字段被复用。当一个页是空闲块的起始页时， `property` 存储该块的 `order` （即 `k`，大小为 `2^k`）。 `PageProperty` 标志位用于表示该页是否为空闲块的头部。

2. 核心函数实现

2.1 初始化函数

初始化 `buddy_area`，将所有链表设置为空，计数器清零。

```
static void buddy_init(void) {
  for (int i = 0; i < MAX_ORDER; i++) {
     list_init(&buddy_area.free_list[i]);
     buddy_area.nr_free[i] = 0;
  }
}
```

2.2 内存映射初始化

这是伙伴系统的初始化，接收一块连续的物理内存， 将 `n` 页内存分解为多个大小为2的幂次方的块。在这里我们采用贪心策略：从 `base` 地址开始，每次都尝试划分出不超过剩余内存 `remaining` 的最大2的幂次方块。对于每个划分出的块，获取其起始页 `current`，设置其 `order`（`current->property = order`），并将其加入到 `buddy_area.free_list[order]` 中。

```
static void buddy_init_mmp(struct Page *base, size_t n) {
    assert(n > 0);
    // 初始化所有页面
    struct Page *p = base;
    for (; p != base + n; p++)
    {
        assert(PageReserved(p));
        p->flags = 0;
        p->property = 0;
        set_page_ref(p, 0);
    }
  // 将内存块按2的幂次分解
  size_t remain = n;
  struct Page *cur = base;
  while (remain > 0) {
     // 找到最大的2的幂次
     size_t order = 0;
     size_t size = 1;
     while (size * 2 <= remain && order < MAX_ORDER - 1) {
       size <<= 1;
       order++;
     }
    // 加入对应order的链表
    cur->property = order;
    SetPageProperty(cur);
    list_add(&buddy_area.free_list[order], &(cur->page_link));
    buddy_area.nr_free[order]++;
    cur += size;
    remain -= size;
  }
}
```

例如：100页内存会被分解为：64页(order=6) + 32页(order=5) + 4页(order=2)

2.3 分配函数

需要分配大小为n的内存块，我们先使用 `log2(n)` 计算满足 `n` 页所需的最小 `order`。接着从 `order` 开始，向上遍历 `free_list`，找到第一个不为空的链表 `free_list[cur_order]`。如果 `cur_order > order`，说明找到的块太大了，我们就需要分裂。这里设置一个 `while` 循环，不断将块一分为二。在每次分裂中，当前块 `page` 的大小从 `2^cur_order` 减半到 `2^(cur_order-1)`。它的伙伴块 `buddy`（地址为 `page + (1 << (cur_order-1))`）被创建出来。将这个新的伙伴块 设置好 `order`，添加到 `free_list[current_order-1]` 中。这个循环直到 `current_order` 等于所需的 `order`。

最后我们从链表中取出大小合适的块，清除其 `PageProperty` 标志，设置其 `property` 为最终的 `order`，并返回。

```
static struct Page *
buddy_alloc_pages(size_t n)
{
    assert(n > 0);
    // 计算需要的order
    size_t order = log2(n);
    if (order >= MAX_ORDER)
    {
        return NULL;
    }

    // 查找合适的空闲块
    size_t cur_order = order;
    while (cur_order < MAX_ORDER &&
list_empty(&buddy_area.free_list[cur_order]))
    {
        cur_order++;
    }

    // 没有找到合适的块
    if (cur_order >= MAX_ORDER)
    {
        return NULL;
    }

    // 从链表中取出一个块
    list_entry_t *le = list_next(&buddy_area.free_list[cur_order]);
    struct Page *page = le2page(le, page_link);
    list_del(&(page->page_link));
    buddy_area.nr_free[cur_order]--;
    ClearPageProperty(page);

    // 如果块太大，需要分裂
    while (cur_order > order)
    {
        cur_order--;

        // 分裂成两个伙伴块，将右边的块加入链表
        struct Page *buddy = page + (1 << cur_order);
        buddy->property = cur_order;
        SetPageProperty(buddy);
        list_add(&buddy_area.free_list[cur_order], &(buddy->page_link));
        buddy_area.nr_free[cur_order]++;
```

```
    }

    // 设置分配的块
    page->property = order;

    return page;
}

```

分配示例:

- 请求5页 → 需要8页(order=3)

- 如果order=3的链表为空，查找order=4(16页)

- 分裂：16页 → 8页 + 8页，取一个8页分配，另一个8页放回链表

2.4 释放函数（合并算法）

如果我们要释放从 某个地址`base` 开始的 `n` 页内存。与分解同理刚开始要根据 `n` 计算出 `order`。之后不断尝试与同样大小的伙伴合并，这里我使用 `get_buddy(base, order)` 函数计算伙伴块的地址。这个函数使用位运算 `page_idx ^ (1 << order)` 来快速定位伙伴。

这时候不能急着合并伙伴块，我们先检查伙伴块是否在物理内存范围内，以及伙伴块是否是空闲块的头部 (通过`PageProperty` 标志），最后我们要保证伙伴块的 `order` 要与当前块相同。

如果以上条件都满足，这时候我们可以合并了。先从伙伴的空闲链表中移除它，再选择两个伙伴中地址较小的一个作为新块的 `base`。然后让`order` 加一进入下一次循环，尝试与新的、更大的伙伴块合并，直到不能合并。

别忘了将最终合并后的块 `base` 设置好 `order` 和 `PageProperty` 标志，加入到 `free_list[order]` 中。

```
static void
buddy_free_pages(struct Page *base, size_t n)
{
    assert(n > 0);

    // 计算order
    size_t order = log2(n);
    if (order >= MAX_ORDER)
    {
        return;
    }

    // 重置页面状态
    struct Page *p = base;
    for (int i = 0; i < (1 << order); i++, p++)
    {
        assert(!PageReserved(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }

    // 合并伙伴块
    while (order < MAX_ORDER - 1)
    {
        // 计算伙伴块的地址

```

```
        struct Page *buddy = get_buddy(base, order);

        // 检查伙伴块是否空闲且大小相同
        if (buddy < pages || buddy >= pages + npage)
        {
            break; // 伙伴块超出范围
        }

        if (!PageProperty(buddy) || buddy->property != order)
        {
            break; // 伙伴块不空闲或大小不匹配
        }

        // 从链表中删除伙伴块
        list_del(&(buddy->page_link));
        buddy_area.nr_free[order]--;
        ClearPageProperty(buddy);

        // 合并：选择地址较小的作为新块的基地址
        if (buddy < base)
        {
            base = buddy;
        }

        order++;
    }

    // 将合并后的块加入链表
    base->property = order;
    SetPageProperty(base);
    list_add(&buddy_area.free_list[order], &(base->page_link));
    buddy_area.nr_free[order]++;
}
```

2.5 伙伴计算：

```
static struct Page *get_buddy(struct Page *page, size_t order) {

  size_t page_idx = page - pages;

  size_t buddy_idx = page_idx ^ (1 << order);  // 异或翻转order位

  return &pages[buddy_idx];

}
```

为什么翻转第 `order` 位就能找到伙伴?

伙伴系统的定义是：两个大小为 `2^k` 的块，如果它们的地址是连续的，并且共同构成一个大小为 `2^(k+1)` 的、地址对齐的块，那么它们就是伙伴。

两个大小为 `2^order` 的伙伴块，它们的起始页号 `page_idx` 在二进制表示下，只有**第 `order` 位**是不同的。因此，通过异或操作 `^ (1 << order)` 来翻转这一位，就可以精确地从一个伙伴的地址计算出另一个伙伴的地址。

测试检测流程

我设计了以下功能测试方案：

1.连续分配1、2、4页的块。验证alloc_pages功能，如果返回非 `NULL` 指针说明功能无误。然后立即释放内存块，恢复环境。

2.分配两个大小为1的块 `p0` 和 `p1`。 使用 `is_buddy(p0, p1, 0)` 检查它们是否是伙伴关系。在释放后，检查 `free_list[0]` 是否为空，`free_list[1]` 中如果出现了一个新的块。说明两个1页的伙伴块已成功合并为一个2页的块。

3.分配一个16页的大块，如果 `alloc_pages` 返回非 `NULL` 指针说明正确。在释放后，我们使用 `buddy_nr_free_pages()` 检查总空闲页数是否恢复正常。

4.连续分配10个1页的块，让内存出现碎片， 再将这10个块全部释放。打印 `free_list` 的统计信息。理想状态是许多小的空闲块已经消失，存在几个大的、合并后的空闲块。

```
static void buddy_test(void)
{

    // 分配和释放
    struct Page *p0, *p1, *p2;

    p0 = alloc_pages(1);
    assert(p0 != NULL);

    p1 = alloc_pages(2);
    assert(p1 != NULL);

    p2 = alloc_pages(4);
    assert(p2 != NULL);

    cprintf("Allocated: p0=%p (1 page), p1=%p (2 pages), p2=%p (4
pages)\n", p0, p1, p2);

    free_pages(p0, 1);
    free_pages(p1, 2);
    free_pages(p2, 4);

    cprintf("Freed all pages\n");

    // 测试合并
    p0 = alloc_pages(1);
    p1 = alloc_pages(1);

    cprintf("Allocated two 1-page blocks: p0=%p, p1=%p\n", p0, p1);

    // 检查是否是伙伴
    if (is_buddy(p0, p1, 0))
    {
        cprintf("p0 and p1 are buddies\n");
    }

    free_pages(p0, 1);
    free_pages(p1, 1);

    // 大块分配
    p0 = alloc_pages(16);
    assert(p0 != NULL);
    cprintf("Allocated 16 pages: p0=%p\n", p0);
```

```
    free_pages(p0, 16);

    // 测试碎片
    struct Page *pages_array[10];
    for (int i = 0; i < 10; i++)
    {
        pages_array[i] = alloc_pages(1);
        assert(pages_array[i] != NULL);
    }
    cprintf("Allocated 10 single pages\n");

    for (int i = 0; i < 10; i++)
    {
        free_pages(pages_array[i], 1);
    }
    cprintf("Freed 10 single pages\n");

    // 打印空闲块统计
    cprintf("\nFree blocks:\n");
    for (int i = 0; i < MAX_ORDER; i++)
    {
        if (buddy_area.nr_free[i] > 0)
        {
            cprintf("Order %d (2^%d=%d pages): %d blocks\n",
                    i, i, 1 << i, buddy_area.nr_free[i]);
        }
    }
    cprintf("Total free pages: %d\n", buddy_nr_free_pages());

}

```

运行测试

1.在 `kern/init/init.c` 的 `pmm_init` 函数中，将 `pmm_manager` 指向 `buddy_pmm_manager`。

```
    const struct pmm_manager *pmm_manager = &buddy_pmm_manager;
```

2.在 `pmm_init` 函数的末尾，调用 `pmm_manager->check()`。

```
    pmm_manager->init_memmap(base, n);
    pmm_manager->check();
```

3.执行以下命令重新编译并运行 uCore：

```
$ make clean
$ make
$ make qemu
```

结果如图：

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc02012a6 (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc0205140 (virtual)
Kernel executable memory footprint: 21KB
memory management: buddy_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
Allocated: p0=0xffffffffc0345fb0 (1 page), p1=0xffffffffc0345e70 (2 pages), p2=0xfffff
fffc0345f10 (4 pages)
Freed all pages
Allocated two 1-page blocks: p0=0xffffffffc0345e70, p1=0xffffffffc0345e98
p0 and p1 are buddies
Allocated 16 pages: p0=0xffffffffc0345bf0
Allocated 10 single pages
Freed 10 single pages

Free blocks statistics:
Order 1 (2^1=2 pages): 1 blocks
Order 3 (2^3=8 pages): 1 blocks
Order 4 (2^4=16 pages): 1 blocks
Order 5 (2^5=32 pages): 1 blocks
Order 7 (2^7=128 pages): 1 blocks
Order 10 (2^10=1024 pages): 31 blocks
Total free pages: 31930
check alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
```

测试结果显示我们伙伴系统成功地完成了基本内存块的**分配**和**释放**（如分配1、2、4页的块），**分裂**（为满足小请求而拆分大块）和**合并**（将释放的相邻伙伴块组合成大块）机制也工作正常。

在一系列碎片化的分配和释放10个单页块后，`Free blocks statistics`显示系统中存在大量高阶的空闲块（如Order 10, 7, 5等），不是一堆零碎的低阶小块。说明我们的合并逻辑非常有效，减少了外部碎片。

扩展练习Challenge：任意大小的内存单元slub分配算法

实验原理和设计思路

SLUB 分配器构建在底层的物理内存管理器（PMM，如 Buddy System）之上，形成了一个**两层内存管理架构**。

对于小于一个页面的小对象分配，如果直接使用 PMM 分配整个页面，很容易出现内部碎片。SLUB 将一个页面（`slab`）切分成多个固定大小的小对象，按需分配。通过预先分配和缓存对象，`kmalloc`和`kfree`的操作通常只是简单的指针操作，避免了昂贵的页面分配和查找过程，也提高了 CPU 缓存的命中率。

```
+-------------------------------------------------+
|                  应用程序/内核代码                |
|          kmalloc(size) | kfree(obj)             |
+-----------------------|-------------------------+
                        V
+-------------------------------------------------+
|                 SLUB 分配器                      |
```

```
|  - 管理不同大小类别的 slub_cache                    |
|  - 在 slab 页面内分配/释放 object                   |
+-----------------------|-----------------------+
                        V (当需要新页面时)
+-----------------------------------------------+
|            物理内存管理器 (PMM)                  |
|    (e.g., Buddy System, Best-Fit)             |
|          alloc_pages() | free_pages()         |
+-----------------------------------------------+
```

数据结构设计

SLUB 的核心是 `slub_cache` 和 `slab` 这两个结构体。

`slub_cache` 是一个特定大小对象的"管理器"。系统会为每种大小的对象（如16字节、32字节、64字节等）创建一个 `slub_cache`。

```c
struct slub_cache {

  size_t object_size;    // 此 cache 管理的对象大小

  size_t objects_per_slab; // 每个 slab 能容纳多少个对象

  list_entry_t slabs_partial; // 部分空闲的 slab 链表

  list_entry_t slabs_full;   // 完全占满的 slab 链表

  unsigned int nr_slabs;   // 此 cache 总共拥有的 slab 数量

};
```

其中 `slabs_partial` 链接所有部分被使用的 slab。`kmalloc` 会优先从这个链表中寻找空间。`slabs_full` 链接所有**已无空闲对象**的 slab。当 `slabs_partial` 中的一个 slab 被用完时，它会被移到这里。

`slab` 本质上就是一个物理页面，其头部包含元数据，其余空间被划分为多个固定大小的对象。

```c
struct slab {
  void *freelist;     // 指向第一个空闲对象的指针 (形成一个链表)
  size_t inuse;      // 已分配的对象数量
  size_t free_count;    // 剩余空闲对象数
  struct Page *page;    // 指向此 slab 对应的 Page 结构体
  list_entry_t slab_link; // 用于链接到 cache 的链表节点
};
```

所有空闲对象通过在对象自身存储下一个空闲对象的地址，形成一个侵入式单向链表。

我们来看一下cache和slabs的关系：

```
caches[i] (for 64-byte objects)

+------------------+

| object_size: 64  |

| slabs_partial ---|---> [Slab A] <--> [Slab C] <--> ...

| slabs_full    ---|---> [Slab B] <--> [Slab D] <--> ...

+------------------+
```

```
[Slab A] 是一个物理页面
+-------------------------------------------------------------------------+
| meta: {freelist -> obj2, inuse: 2, ...} | obj1 | obj2 | obj3 | ... |
+---------------------------------------- |------|------|------|-----+
                                          |(used)|  |   |(used)|
                                                V   |
                                              [obj4]<-+
                                                |
                                                V
                                               NULL
```

在上图中，`Slab A` 是一个部分使用的 slab。它的 `freelist` 指向 `obj2`，而 `obj2` 的内存空间里存储着下一个空闲对象 `obj4` 的地址，`obj4` 指向 `NULL`，表示空闲链表结束。

代码编写步骤

1. `slub_init()` 初始化函数

```c
void slub_init(void)
{
    cprintf("Initializing Slabs\n");

    for (int i = 0; i < SLUB_SIZE_CLASSES; i++)
    {
        caches[i].object_size = size_classes[i];

        // 计算每个slab可容纳的对象数
        size_t slab_overhead = ROUNDUP(sizeof(struct slab), 8);
        size_t available = PGSIZE - slab_overhead;
        caches[i].objects_per_slab = available / size_classes[i];

        list_init(&caches[i].slabs_partial);
        list_init(&caches[i].slabs_full);
        caches[i].nr_slabs = 0;

        cprintf("  Size class %d: object_size=%d, objects_per_slab=%d\n",
                i, size_classes[i], caches[i].objects_per_slab);
    }
}
```

遍历预设的 `size_classes` 数组，为每个大小类别初始化一个 `caches[i]` 结构体。用一个页面 `PGSIZE` 减去 `sizeof(struct slab)` 的元数据开销，再除以对象大小，就得到了每个 slab 能容纳的对象数量。最后初始化 `slabs_partial` 和 `slabs_full` 链表。

2. `kmalloc(size_t size)`

```
void *kmalloc(size_t size)
{
    if (size == 0 || size > SLUB_MAX_SIZE)
    {
        return NULL;
    }

    // 找到合适的大小类别
    int idx = get_size_class_index(size);
    if (idx < 0)
    {
        // 太大，直接分配页面
        size_t pages = ROUNDUP(size, PGSIZE) / PGSIZE;
        struct Page *page = alloc_pages(pages);
        return page ? page2kva(page) : NULL;
    }

    struct slub_cache *cache = &caches[idx];
    struct slab *slab = NULL;

    // 1. 尝试从partial链表获取slab
    if (!list_empty(&cache->slabs_partial))
    {
        list_entry_t *le = list_next(&cache->slabs_partial);
        slab = to_struct(le, struct slab, slab_link);
    }
    // 2. 创建新的slab
    else
    {
        slab = slub_create_slab(cache);
        if (slab == NULL)
        {
            return NULL;
        }
        list_add(&cache->slabs_partial, &slab->slab_link);
        cache->nr_slabs++;
    }

    // 从freelist获取对象
    void *object = slab->freelist;
    if (object == NULL)
    {
        cprintf("Error: freelist is NULL but slab is in partial list!\n");
        return NULL;
    }

    slab->freelist = *(void **)object;
    slab->inuse++;
    slab->free_count--;

    // 如果slab已满，移动到full链表
```

```
    if (slab->free_count == 0)
    {
        list_del(&slab->slab_link);
        list_add(&cache->slabs_full, &slab->slab_link);
    }

    // 清零对象内存
    memset(object, 0, cache->object_size);

    return object;
}
```

我们先根据请求的 `size`，通过 `get_size_class_index` 找到最合适的 `slub_cache`。优先从 `cache->slabs_partial` 链表中取出一个 slab。 如果 `slabs_partial` 为空，则调用 `slub_create_slab` 创建一个***新的 slab。**

创建新 Slab 时先调用 `alloc_page()` 从底层 PMM 获取一个新页面。初始化 `struct slab` 元数据后遍历页面中所有对象的存储空间，将它们像串珠子一样链接起来，形成初始的空闲对象链表。将这个新创建的 slab 添加到 `slabs_partial` 链表中。

接着分配对象，从所选 slab 的 `freelist` 中"弹出"第一个空闲对象，更新 `freelist` 指向下一个空闲对象。最后更新 slab 的 `inuse` 和 `free_count` 计数。

3.`kfree(void \*obj)`

```
void kfree(void *obj)
{
    if (obj == NULL)
    {
        return;
    }

    // 获取对象所属的页面
    struct Page *page = addr_to_page(obj);
    struct slab *slab = (struct slab *)page2kva(page);

    // 找到对应的cache
    struct slub_cache *cache = NULL;
    for (int i = 0; i < SLUB_SIZE_CLASSES; i++)
    {
        if (caches[i].object_size >= ((char *)obj - (char *)slab))
        {
            // 简化判断：通过偏移粗略确定
            cache = &caches[i];
            break;
        }
    }

    if (cache == NULL)
    {
        // 可能是大对象，直接释放页面
        free_page(page);
        return;
    }

    // 判断slab之前的状态
    bool was_full = (slab->free_count == 0);
```

```
    // 将对象加入freelist
    *(void **)obj = slab->freelist;
    slab->freelist = obj;
    slab->inuse--;
    slab->free_count++;

    // 根据使用情况调整链表
    if (slab->free_count == cache->objects_per_slab)
    {
        // 完全空闲，释放slab
        list_del(&slab->slab_link);
        free_page(page);
        cache->nr_slabs--;
    }
    else if (was_full)
    {
        // 从full变为partial
        list_del(&slab->slab_link);
        list_add(&cache->slabs_partial, &slab->slab_link);
    }
}
```

`addr_to_page(obj)` 函数可以从任意一个对象地址反向定位到它所在的物理页面，从而找到 `struct slab` 的元数据，再根据对象大小（或通过其他机制）找到对应的 `slub_cache`。

我们将被释放的对象 `obj` “压入”slab 的 `freelist` 头部，更新 slab 的 `inuse` 和 `free_count` 计数。

如果 slab 之前是满的（`was_full`），现在因为释放而有了空位，则将其从 `slabs_full` 链表移回到 `slabs_partial` 链表，以便下次分配时重用。如果 slab 在释放后完全空闲（`inuse == 0`），则直接调用 `free_page()` 将整个页面归还给底层 PMM，以回收内存。

测试思路与过程

1.测试不同大小的对象能否被正确分配和释放。我们连续为每个主要的大小类别分配一个对象，验证所有 `slub_cache` 都能正常工作，并能成功创建和使用新的 slab。

```
    void *obj1 = kmalloc(16);
    void *obj2 = kmalloc(32);
    void *obj3 = kmalloc(64);
    void *obj4 = kmalloc(128);

    assert(obj1 != NULL && obj2 != NULL && obj3 != NULL && obj4 != NULL);
    cprintf("  Allocated: obj1=%p, obj2=%p, obj3=%p, obj4=%p\n",
            obj1, obj2, obj3, obj4);
```

2.我们向之前分配的对象中写入数据 （字符串和整数),验证返回的内存地址有效且可用。

```
    strcpy((char *)obj1, "Hello");
    *(int *)obj2 = 12345;
    cprintf("  obj1='%s', obj2=%d\n", (char *)obj1, *(int *)obj2);
```

3.释放所有对象。

```
kfree(obj1);
kfree(obj2);
kfree(obj3);
kfree(obj4);
```

4.在一个循环中分配大量（如20个）相同大小的对象，然后全部释放。强制 slab 从 `partial` 状态转移到 `full` 状态，然后再因为释放而移回 `partial`，甚至可能因为完全空闲而被回收，测试鲁棒性。

```
#define BATCH_SIZE 20
    void *objs[BATCH_SIZE];
    for (int i = 0; i < BATCH_SIZE; i++)
    {
        objs[i] = kmalloc(64);
        assert(objs[i] != NULL);
    }
    cprintf("  Allocated %d objects\n", BATCH_SIZE);

    for (int i = 0; i < BATCH_SIZE; i++)
    {
        kfree(objs[i]);
    }
    cprintf("  Freed %d objects\n", BATCH_SIZE);
```

5.测试碎片。分配一组对象，然后交错地释放其中一部分，再重新分配，最后全部释放。

```
void *small[10];
for (int i = 0; i < 10; i++)
{
    small[i] = kmalloc(16);
}

// 释放一半
for (int i = 0; i < 5; i++)
{
    kfree(small[i * 2]);
}

// 再分配
for (int i = 0; i < 5; i++)
{
    small[i * 2] = kmalloc(16);
}

// 全部释放
for (int i = 0; i < 10; i++)
{
    kfree(small[i]);
}
cprintf("  释放完所有slabs\n");

for (int i = 0; i < SLUB_SIZE_CLASSES; i++)
{
    if (caches[i].nr_slabs > 0)
    {
        cprintf("  Size %d: %d slabs\n",
                size_classes[i], caches[i].nr_slabs);
```

```
    }
}

```

测试结果

我们设计的SLUB 依赖于 PMM，我把PMM配置为 `buddy_pmm_manager`，因为 Buddy System 能高效地提供页面。

执行下列命令编译执行：

```
$ make clean
$ make
$ make qemu
```

输出结果如下：

```
Order 7 (2^7=128 pages): 1 blocks
Order 10 (2^10=1024 pages): 31 blocks
Total free pages: 31929
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
Initializing Slabs
  Size class 0: object_size=16, objects_per_slab=253
  Size class 1: object_size=32, objects_per_slab=126
  Size class 2: object_size=64, objects_per_slab=63
  Size class 3: object_size=128, objects_per_slab=31
  Size class 4: object_size=256, objects_per_slab=15
  Size class 5: object_size=512, objects_per_slab=7
  Size class 6: object_size=1024, objects_per_slab=3

SLUB Test:
  Allocated: obj1=0xffffffffc7fe7030, obj2=0xffffffffc7ff7030, obj3=0xffffffffc7fe8030
, obj4=0xffffffffc7fe9030
  obj1='Hello', obj2=12345
Free objects
 freed
  Allocated 20 objects
  Freed 20 objects
  Fragmentation test passed

SLUB Statistics:
  Size 16: 1 slabs
  Size 32: 1 slabs
  Size 64: 1 slabs
  Size 128: 1 slabs
```

输出结果显示我们成功分配了四个不同大小的对象，且可以正常读写，可以正常被释放。连续分配20个64字节的对象也无误，没有出现内存泄漏现象。在内存碎片化情况下SLUB仍然能正确重用之前释放的对象空间。freelist链表管理逻辑没有问题。

Challenge3: 硬件的可用物理内存范围的获取方法

问题：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?

这个问题可以转化为：**如何让 OS 找到能够自由刷写数据的 DRAM 区域，需要知道开始地址和结束地址。** 至于后续，OS 内核所占空间一定是预先规定好的，而内核后内核代码和数据段等占用的空间在 `kernel.ld` 中定义了 `end` 来获取这一切结束的虚拟地址，按规则转化为物理地址即可。开始地址加上得到的这个物理地址，到原来的结束地址这个区间内就是实际可用的物理内存范围。

问题的核心就转变为了找到可用 DRAM 的开始地址和结束地址。

(1) 除了指导书中提到的 bootloader —— qemu 中是 OpenSBI 来完成对于物理内存的扫描，告诉 OS 哪些地址范围是可用的 DRAM. 除了 bootloader 之外，还有 x86 的 BIOS、UEFI 固件的 GetMemoryMap() 接口等通过类似的方法获得整体的物理内存布局，从而获取可用 DRAM 的地址范围。

(2) 但本质上，上面的这些方法都是有软/硬件提前告诉 OS 可用的内存范围的。如果这些方法都被禁用的情况下，或许只能够通过逐块访问物理内存内测试它是否允许使用，从而确定范围。几个步骤：

- **首先确定起始点。** 如果在链接文件中写的是实际物理地址，或者虽然是虚拟地址但已经敲定了虚拟地址到物理地址的变换公式，亦或是在 make 的时候直接指定了加载地址，就可以从这些信息出发直接从内核摆放的末尾地址开始向后扫描。但一般正式的系统设计像不会这么做，那就能从头开始扫描。
- **确定扫描单位。** 还是选择以 4KB 的页为单位较妥当，但对于一般物理内存结构是前面有一大部分不可用的规律来看，可以刚开始采用较大的单位，而后慢慢以超过线性速度的递减连续划分直到 4KB.
- **单位内测试方法。** 课上曾经提到过，可以写入 `0xA5A5A5A5A5A5A5A5`，`0x5A5A5A5A5A5A5A5A`，甚至全0和全1，再读出，来测试改块内存的可用性。多几个测试数据是为了降低误判。
- **边界判断机制。** 使用异常处理捕获机制，一旦引发异常直接跳出进行下一块的检测，或者说明已经到达物理内存尾部（根据硬件参数提前确定遍历测试的最大范围并设置最高访问地址，可以避免一直向后访问不存在的地址）。由于一般情况下要从头开始扫描，一开始也是不可使用的，那么可以设置 flag 等机制，将 “不可用-可用-不可用” 的变化地址记录下来，中间可用的部分地址就可以求出。
- 整个过程通过 CPU 直接写指令到内存执行，其实在可能触发异常损害 CPU 相关硬件（MMIO/寄存器）这件事上，这个操作方案并不太安全。

(3) 搜索网络了解到，可以利用外设来进行更安全的检测。其中一个是 DMA，它通过配置一个 DMA 描述符/Descripitor 来将内存中一个设置好的 buf1 区内的源数据写入目标物理页，而后 DMA 将该物理页的数据读出存放在 buf2 区，CPU 再读取 buf2 的值和 buf1 对比，一致则说明是可用 DRAM. 其安全性来源于，整个访问（读+写）未知物理页的过程 CPU 均不参与，因为 DMA engine 独立于 CPU，这可以减少破坏内核/设备寄存器的风险。可惜这个方法仍然 **需要知道一段已知可用内存**（安全页），来存放测试用的源数据。

```
DMA描述符是什么？
DMA 通常使用一个结构体描述一次传输：
(源地址，目标地址，传输长度，控制标志)
内核需要在早期初始化这个 DMA descriptor 并写入设备寄存器。
```

重要知识点总结

1. OS原理对应的知识点

实验知识点	OS原理知识点	关系说明
First-Fit算法	动态分区分配	实验实现了教材中的经典首次适应算法
Best-Fit算法	动态分区分配	通过完整遍历找最优解，减少外部碎片
Buddy System	伙伴系统分配	2的幂次方分配，支持快速分裂与合并
SLUB分配器	Slab分配器	小对象内存管理，减少内部碎片
空闲链表管理	空闲块管理	用双向链表维护不同大小的空闲块
页面合并	碎片整理	释放时的即时合并策略（连续物理地址）
伙伴合并	碎片整理	通过伙伴关系递归合并，恢复大块内存
Page结构体	页框管理	操作系统对物理页面的抽象描述
分层内存管理	内存管理层次	PMM(页级) → SLUB(对象级) 的两层架构
物理内存探测	启动时内存初始化	通过DTB/BIOS获取可用内存范围

2. 实验中的重要概念

2.1 基础内存管理概念

- **property字段**：在不同算法中复用，记录空闲块大小(First/Best-Fit)或order值(Buddy)
- **PG_property标志**：标识该页是否为空闲块首页
- **页面合并**：释放时自动合并相邻空闲块，减少外部碎片
- **地址有序**：First/Best-Fit链表按物理地址排序，便于合并操作

2.2 Buddy System核心概念

- **Order值**：块大小为2^order个页面，便于快速计算和对齐
- **伙伴关系**：两个大小相同、地址相邻且满足对齐条件的块互为伙伴
- **伙伴计算**：通过异或操作 `page_idx ^ (1 << order)` 快速定位伙伴块
- **分裂操作**：大块一分为二，一个分配/继续分裂，另一个加入空闲链表
- **递归合并**：释放时递归检查伙伴状态，最大化合并空闲块
- **分级链表**：维护MAX_ORDER个链表，每个链表管理固定大小的空闲块

2.3 SLUB分配器核心概念

- **slub_cache结构**：管理特定大小对象的缓存池，每种大小一个cache
- **slab结构**：一个物理页面被划分为多个固定大小的对象
- **freelist机制**：侵入式单向链表，在空闲对象自身存储下一个空闲对象地址
- **size_classes**：预定义大小类别(16, 32, 64, 128, 256, 512, 1024, 2048字节)
- **slabs_partial/full链表**：partial存放部分使用的slab，full存放已满的slab
- **两层架构**：SLUB构建在PMM之上，小对象用SLUB，大对象直接用PMM
- **kmalloc/kfree接口**：提供类似malloc/free的动态内存分配接口

2.4 物理内存初始化

- **DTB解析**：从设备树获取物理内存基地址和大小
- **页表初始化**：建立pages数组，每个物理页对应一个Page结构
- **内存映射**：计算va_pa_offset，实现虚拟地址与物理地址转换
- **page2kva/page2pa**：页面到虚拟地址/物理地址的转换函数

3. 实现难点与收获

3.1 First-Fit 与 Best-Fit 难点

- 理解链表操作宏（`le2page`, `list_add_before`等）
- 正确处理块分割和合并的边界情况
- Best-Fit算法需要完整遍历链表，不能提前退出
- 维护链表地址有序性，确保合并操作正确

收获:

- 深入理解了物理内存管理的底层实现
- 掌握了First-Fit和Best-Fit算法的时间复杂度权衡
- 学会了如何使用双向链表管理动态数据结构
- 理解了外部碎片的形成原因和缓解方法

3.2 Buddy System 难点

- 理解2的幂次方分配的数学原理
- 使用位运算快速计算伙伴地址（异或操作）
- 处理分裂时的递归逻辑，从大块到小块
- 处理合并时的递归逻辑，检查伙伴是否空闲
- 维护多个不同order的空闲链表

收获:

- 掌握了伙伴系统的分裂与合并算法
- 理解了2的幂次方分配的优势（快速计算、对齐）
- 学会了使用位运算优化地址计算
- 理解了如何平衡内部碎片和外部碎片

3.3 SLUB分配器难点

- 理解两层内存管理架构（PMM + SLUB）
- 实现侵入式freelist链表（在对象内存中存储指针）
- 正确计算每个slab能容纳的对象数量（考虑元数据开销）
- 实现slab在partial和full链表之间的动态转移
- 从对象地址反向定位到所属slab和cache（addr_to_page）
- 处理完全空闲slab的回收时机

收获:

- 深入理解了Linux内核的slab/slub分配器原理
- 掌握了小对象内存管理的高效方法
- 学会了如何减少内部碎片和提高缓存命中率
- 理解了对象缓存池的设计思想

3.4 物理内存探测难点

- 理解设备树（DTB）的结构和解析方法
- 处理不同硬件平台的内存布局差异
- 理解MMIO、保留内存等特殊区域
- 实现安全的内存探测方法（避免访问非法地址）

收获:

- 了解了操作系统启动时的内存初始化流程
- 掌握了多种获取物理内存范围的方法
- 理解了DMA等外设在内存探测中的应用

4. 算法对比与选择

算法	时间复杂度(分配)	内部碎片	外部碎片	适用场景
First-Fit	O(n) 平均较快	较大	较多	快速分配，对碎片不敏感
Best-Fit	O(n) 必须遍历	较小	较少	需要精确匹配，减少浪费
Buddy System	O(log n)	较多(2^n对齐)	很少	频繁分配释放，需要快速合并
SLUB	O(1) 平均	很少	无(固定大小)	小对象频繁分配释放

5. 实验代码优化建议

1. **First/Best-Fit优化**：
   - 封装重复的链表插入逻辑为独立函数
   - 添加自旋锁保证多核环境下的线程安全
   - Best-Fit可改为按大小排序链表，优化为O(1)查找
2. **Buddy System优化**：
   - 使用位图快速标记块的占用状态
   - 实现延迟合并策略，减少频繁的链表操作
   - 支持更大的MAX_ORDER以适应大内存系统
3. **SLUB优化**：
   - 实现per-CPU缓存，减少锁竞争
   - 添加slab着色（coloring）提高缓存性能
   - 实现slab的预分配和批量释放

OS原理中重要但实验未涉及的知识点

1. **分页与分段结合**：实验仅实现了页式管理，未涉及段式或段页式
2. **多级页表**：实验使用简单的页面管理，未实现多级页表结构
3. **虚拟内存**：实验专注于物理内存，虚拟内存管理在后续实验
4. **页面置换算法**：LRU、Clock、LFU等算法未在本实验中体现
5. **内存保护**：权限控制、越界检查、写时复制等安全机制
6. **NUMA架构**：现代多核系统的内存访问优化和亲和性管理
7. **内存压缩**：zswap、zRAM等内存压缩技术
8. **大页支持**：HugePage、Transparent HugePage的实现
9. **内存回收**：kswapd、OOM killer等内存回收机制
10. **内存热插拔**：运行时动态增删物理内存