OS lab2

成员信息

解子萱 2312585 信息安全、法学双学位班

崔颖欣 2311136 信息安全、法学双学位班

范鼎辉 2312326 信息安全

练习1：理解first-fit 连续物理内存分配算法（思考题）

> first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法，需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合 `kern/mm/default_pmm.c` 中的相关代码，认真分析default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。 请在实验报告中简要说明你的设计实现过程。请回答如下问题：
>
> - 你的first fit算法是否有进一步的改进空间?

准备工作

在开始解析first-fit 连续物理内存分配算法之前，有必要对一些关键头文件进行整理，理解内存的基本实现结构和基本原理设计。

空闲页链表的实现：通过list.h 定义环形双向链表结构

基本双向链表实现如下：

结构体/函数	代码实现	功能解析

`list_entry` 节点结构体	`struct list_entry {`
	`struct list_entry *prev, *next;`
	`};`
		• 每个节点的 `prev` 指针指向前一个节点；`next` 指针指向下一个节点
		• `typedef` 定义 `list_entry_t` 为 `struct list_entry` 的别名

`list_init` 链表初始化	`static inline void list_init(list_entry_t *elm) {`	• 传入链表节点，将该节点的 `prev` 和 `next` 都指向自己，作为孤立的表头

	`elm->prev = elm->next = elm;` `}`	

`__list_add` 节点插入的底层函数	`static inline void __list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t *next) {`
	`prev->next = next->prev = elm;`
	`elm->next = next;`
	`elm->prev = prev;`
	`}`
		• 将新节点 `elm` 插入到已知的前一个节点 `prev` 和下一个节点 `next` 之间，并调整它们的 `next` 和 `prev` 指针。

`__list_del` 节点删除的底层函数	`static inline void __list_del(list_entry_t *prev, list_entry_t *next) {`
	`prev->next = next;`
	`next->prev = prev;`
	`}`
		• 删除节点 `prev` 和 `next` 之间的节点。“前的后为后，后的前为前”，跳过中间的节点，实现删除操作。

`list_empty` 检查链表是否为空	`static inline bool list_empty(list_entry_t *list) {return list->next == list;` `}`	• 如果链表为空，则 `list->next` 会指向 `list` 本身，即初始化时的状态（孤立表头）
`list_next` 获取下一个节点	`static inline list_entry_t *list_next(list_entry_t *listelm) {return listelm->next;` `}`	• 通过 `listelm->next` 获取下一个节点的指针。
`list_prev` 获取上一个节点	`static inline list_entry_t *list_prev(list_entry_t *listelm) {return listelm->prev;` `}`	• 通过 `listelm->prev` 获取前一个节点的指针。

在此基础上，一些包装后的上层函数包括：

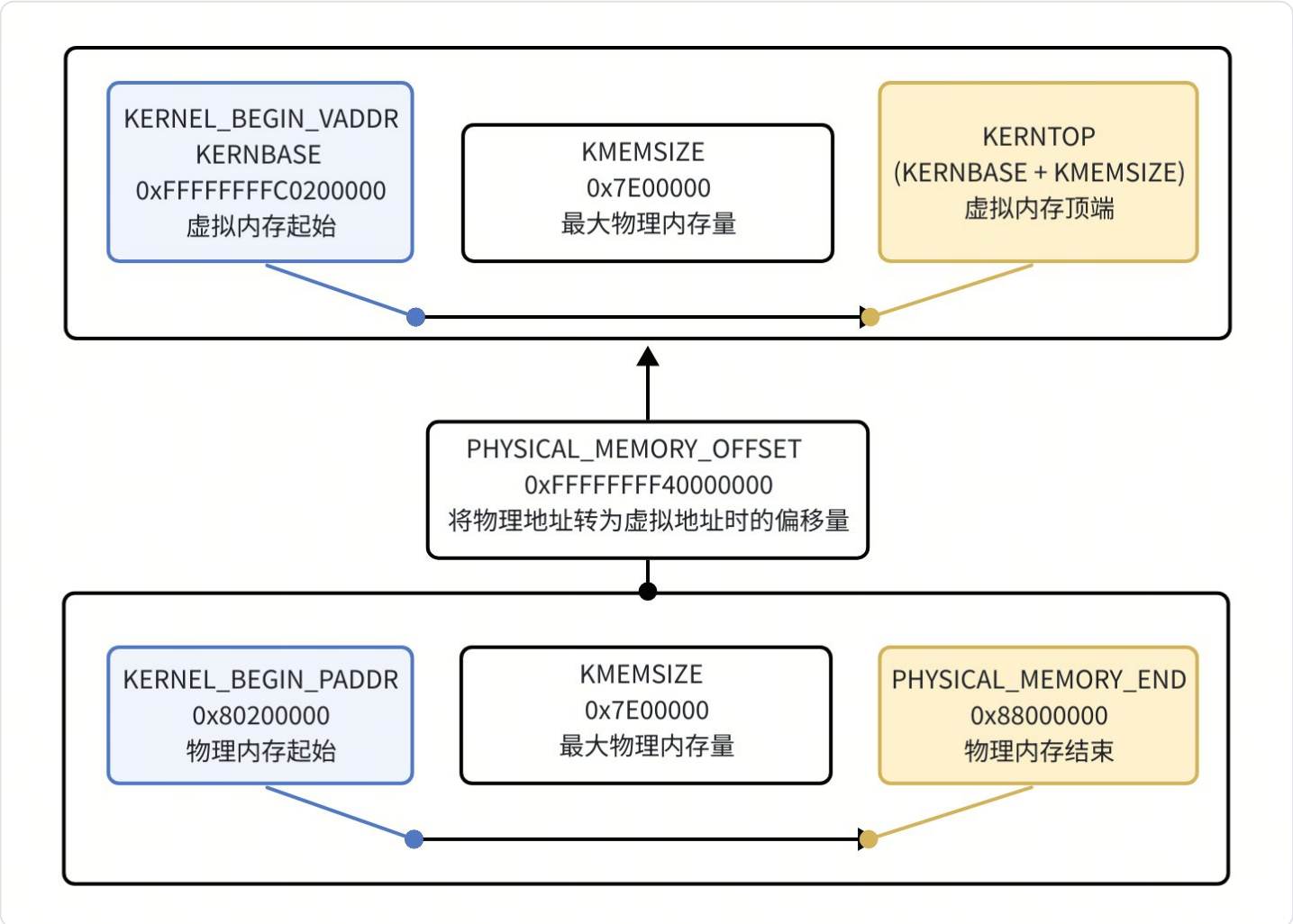
- `list_add` 节点插入封装（默认向后）：调用 `list_add_after`，将新节点 `elm` 添加到旧节点 `listelm` 之后
- `list_add_before` 节点向前插入的封装：调用 `__list_add`，将新节点 `elm` 在 `listelm` 节点之前插入。
- `list_add_after` 节点向后插入的封装：调用 `__list_add`，将新节点 `elm` 在 `listelm` 节点之后插入。

- `list_del` 节点删除封装：调用 `__list_del`，将 `listelm` 节点从链表中移除
- `list_del_init` 节点删除+初始化封装：先调用 `list_del` 删除节点，再调用 `list_init` 初始化节点，使其成为一个孤立的节点。

内存结构实现：通过memlayout.h 定义内存布局结构

顶部地址映射宏定义

关键示意图总结如下：



其中0x80000000到0x80200000被OpenSBI占用

栈页大小宏定义

主要涉及下面两句代码：

```
#define KSTACKPAGE           2
#define KSTACKSIZE           (KSTACKPAGE * PGSIZE)
```

- 意味着内核栈由 2 页组成，栈大小为2*页大小

Page页结构定义

根据代码定义，一个页结构拥有以下属性：

- `ref` 引用计数， `ref == 0` 表示当前没有引用
- `flags` 标志位，用来表示页面的各种状态，两个标志位（bit）：

名称	含义	位编号
PG_reserved	页被内核保留，用户分配器不能使用；若 PG_reserved = 1，代表这个页不可分配；若 PG_reserved = 0，代表这个页可以被分配或管理。	第 0 位（最低位）
PG_property	页是空闲块的头页（管理连续空闲页块）；如果一个空闲块包含连续的多页，只有头页会设置 PG_property;	第 1 位

- `property` 连续空闲块的页数：当一个连续空闲区以某页为头时，该页的 `property` 等于该连续空闲块的页数，而非头页则把 `property` 设为 0。
- `page_link` 一个 `list_entry_t` 链表节点，用于将此 `Page` 节点插入到空闲链表 `free_list` 中

特殊宏——le2page 宏：已知 `Page` 中的 `page_link` 字段的地址，把它转回到对应的 `Page` 指针

Free_area空闲区结构定义

- `free_area_t` 作为struct结构，其实是封装保存一个空闲页链表的头节点（ `free_list` ）和该链表中页的总数（ `nr_free` ）。

default_pmm.c 代码分析

default_init 初始化工作

我们首先针对全局内存空区进行了一系列初始化准备工作：

代码块

```
//内存空闲区定义，
static free_area_t free_area;
#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)
//初始化函数，初始化空页链表的头节点，设置总节点（空页表）数量为0：
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
//获取空页表数量的函数
```

```
12  static size_t
13  default_nr_free_pages(void) {
14      return nr_free;
15  }
```

default_init_memmap 空闲页初始化工作

这一部分我们将连续的物理页（从 base 开始共 n 个）初始化为可分配的空闲页：

代码块

```
1   static void
2   default_init_memmap(struct Page *base, size_t n) {
3       assert(n > 0);
4       struct Page *p = base;
5       //遍历每一页，确保：1.都是保留页 2.清零页属性 3.清零页引用计数
6       for (; p != base + n; p ++) {
7           assert(PageReserved(p));
8           p->flags = p->property = 0;
9           set_page_ref(p, 0);
10      }
11      //设置头页“base”的属性，并把这 n 页加入空闲总量统计
12      base->property = n;
13      SetPageProperty(base);
14      nr_free += n;
15      //如果当前空闲链表为空，就直接把这块插入链表头后面
16      if (list_empty(&free_list)) {
17          list_add(&free_list, &(base->page_link));
18      }
19      //否则，遍历整个空闲链表，寻找合适的插入位置：
20      else {
21          list_entry_t* le = &free_list;
22          while ((le = list_next(le)) != &free_list) {
23              struct Page* page = le2page(le, page_link);
24              //如果 base < page：说明新块地址更靠前，按地址顺序插入到 page 前面；
25              if (base < page) {
26                  list_add_before(le, &(base->page_link));
27                  break;
28              }
29              //如果已经到达链表尾，表示新块地址比所有已有块都大，把它加在链表尾部。
30              else if (list_next(le) == &free_list) {
31                  list_add(le, &(base->page_link));
32              }
33          }
34      }
35  }
```

default_alloc_pages 内存页 First Fit 分配算法

其主体思想是：在空闲页链表 free_list 中找到第一块连续的空闲页，并从中划出 n 页供使用。

代码块

```
static struct Page *
default_alloc_pages(size_t n) {
    //大小检查
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    //遍历空闲链表，寻找合适的空闲页page
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        // 关键点：p->property 表示该空闲块的页数，找到第一个 property >= n 的块，即 First Fit;
        // 若找到，则保存到 page 并 break。
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    // 如果找到了合适的page，进行以下操作：
    if (page != NULL) {
        //先取出当前块的前驱结点 prev，再将该空闲块从空闲链表中删除（因为要分配它）；
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        //处理分配后的剩余部分
        if (page->property > n) {
            struct Page *p = page + n;        //新的空闲块头页 = page + n;
            p->property = page->property - n; //它的 property = 原property - n;
            SetPageProperty(p);               //设置 PG_property 位;
            list_add(prev, &(p->page_link));  //把它插在原空闲块的前驱后面，保持地址顺序
        }
        // 减少系统空闲页计数；并清除 PG_property，标记这块页已不再是空闲块;
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

default_free_pages 物理页释放算法

其主要思想是：释放从 base 开始的 n 个连续页Page ，并将它们加入系统的空闲页链表 free_list 中

代码块

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    //对每一页检查它：1.不是保留页 2.不属于已标记的空闲块;
    //随后清空标志位与引用计数，防止被错误认为仍在使用
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    //设定空闲块头信息
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    //将获得的空闲页插入 free_list ，并保证按物理地址升序排列，逻辑与初始化时一致，故不再重复
    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));
            }
        }
    }
    // 空闲页向前合并的步骤
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        // 看链表中 base 前一个空闲块 p 是否紧邻
        //条件 p + p->property == base表示物理上连续，若相邻则合并:
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            //关键点：更新新的块头是更前面的页
```

```
            base = p;
        }
    }
    // 空闲页向后合并的步骤，与上一致，不再重复
    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));
        }
    }
}
```

check 内存设置自检测函数

basic_check代码及检测总结如下：

代码块

```
static void
basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    assert(alloc_page() == NULL);

    free_page(p0);
    free_page(p1);
```

```
27      free_page(p2);
28      assert(nr_free == 3);
29
30      assert((p0 = alloc_page()) != NULL);
31      assert((p1 = alloc_page()) != NULL);
32      assert((p2 = alloc_page()) != NULL);
33
34      assert(alloc_page() == NULL);
35
36      free_page(p0);
37      assert(!list_empty(&free_list));
38
39      struct Page *p;
40      assert((p = alloc_page()) == p0);
41      assert(alloc_page() == NULL);
42
43      assert(nr_free == 0);
44      free_list = free_list_store;
45      nr_free = nr_free_store;
46
47      free_page(p);
48      free_page(p1);
49      free_page(p2);
50  }
```

阶段	动作	检查点	意图
1	分配三页	页不重复、ref=0	验证 alloc 基本正确性
2	检查物理地址	地址在合法范围内	验证 page→pa 映射
3	清空空闲链表	alloc 应失败	测试极端条件
4	释放页后再分配	成功分配3页	检查 free/alloc 互通
5	释放单页再分配	返回相同页	验证链表与计数逻辑
6	恢复链表状态	无副作用	保持全局一致性
7	释放残留页	内存清理	防止污染全局环境

default_check代码及检测总结如下：

代码块

```
1   static void
2   default_check(void) {
3       int count = 0, total = 0;
4       list_entry_t *le = &free_list;
5       while ((le = list_next(le)) != &free_list) {
6           struct Page *p = le2page(le, page_link);
```

```
        assert(PageProperty(p));
        count ++, total += p->property;
    }
    assert(total == nr_free_pages());

    basic_check();

    struct Page *p0 = alloc_pages(5), *p1, *p2;
    assert(p0 != NULL);
    assert(!PageProperty(p0));

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));
    assert(alloc_page() == NULL);

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    free_pages(p0 + 2, 3);
    assert(alloc_pages(4) == NULL);
    assert(PageProperty(p0 + 2) && p0[2].property == 3);
    assert((p1 = alloc_pages(3)) != NULL);
    assert(alloc_page() == NULL);
    assert(p0 + 2 == p1);

    p2 = p0 + 1;
    free_page(p0);
    free_pages(p1, 3);
    assert(PageProperty(p0) && p0->property == 1);
    assert(PageProperty(p1) && p1->property == 3);

    assert((p0 = alloc_page()) == p2 - 1);
    free_page(p0);
    assert((p0 = alloc_pages(2)) == p2 + 1);

    free_pages(p0, 2);
    free_page(p2);

    assert((p0 = alloc_pages(5)) != NULL);
    assert(alloc_page() == NULL);

    assert(nr_free == 0);
    nr_free = nr_free_store;

    free_list = free_list_store;
    free_pages(p0, 5);
```

```
54
55      le = &free_list;
56      while ((le = list_next(le)) != &free_list) {
57          struct Page *p = le2page(le, page_link);
58          count --, total -= p->property;
59      }
60      assert(count == 0);
61      assert(total == 0);
62  }
```

阶段	检查项目	验证点
1	链表遍历	所有空闲块有 PageProperty
2	全局计数	nr_free_pages() 与链表统计一致
3	分配单块	能成功分配连续页块
4	分配失败	无空闲页时 alloc_page() 返回 NULL
5	部分释放	释放中间页段后能正确形成空闲块
6	块拆分	分配部分页后剩余块保留正确 property
7	块合并	释放相邻块后能自动合并
8	链表有序	分配时按地址顺序维护
9	计数一致性	所有操作后 count/total 不变

default_pmm_manager物理内存分配器初始化

这个结构体本质上是一个函数指针表，描述该物理内存管理器应当实现的所有接口函数，相当于对上述所有工作做整合封装。

代码块

```
1  const struct pmm_manager default_pmm_manager = {
2      .name = "default_pmm_manager",      // 初始化内存管理器的名字
3      .init = default_init,               // 初始化函数（如初始化链表、锁等）
4      .init_memmap = default_init_memmap,// 初始化内存页结构（把物理页标记为空闲）
5      .alloc_pages = default_alloc_pages,// 分配指定数量的物理页
6      .free_pages = default_free_pages,  // 释放物理页
7      .nr_free_pages = default_nr_free_pages, // 查询当前空闲页数
8      .check = default_check,             // 检查函数，用于调试或验证正确性
9  };
```

程序物理内存分配过程

下面总结 `default_pmm.c` 在整个物理内存分配中的作用：

- `default_init` 初始化整个全局空页链表，并设置总节点数量为0；
- `default_init_memmap` 将连续的n个物理页初始化为可分配的空闲页，并插入空闲链表；
- `default_alloc_pages` 实现FirstFit内存分配算法，在空闲页链表中找到第一块连续的空闲页，并从中划出 n 页供使用；
- `default_free_pages` 释放 n 个连续的内存页 ，并将它们加入空闲页链表中，随后完成空页上下文合并。

总的来说，系统启动时， `default_init` 建立物理页管理的基础结构；`default_init_memmap` 形成可供分配的内存池。当内核或程序请求分配内存时，`default_alloc_pages` 按照First Fit算法将所需页划出并标记为已用；当内存被释放时，`default_free_pages` 负责将这些页重新插入空闲链表，并自动与相邻空闲块合并。整个流程实现了物理页从初始化、分配到回收的完整管理机制。

算法改进空间

1. 对于free_list的遍历性能优化

当前 `default_init_memmap` 、 `default_alloc_pages` 和 `default_free_pages` 都是从头到尾对空闲页链表进行遍历：

代码块

```
list_entry_t* le = &free_list;
while ((le = list_next(le)) != &free_list) {
    ...
}
```

很明显这是线性 O(n) 查找插入点，如果空闲页块较多，每次释放或初始化都会线性扫描，性能会急剧下降！因此可以引入一个简单的按地址有序插入二叉搜索树，来把释放或插入复杂度降为 O(log n)。

2. 空闲页合并逻辑优化

在 `default_free_pages()` 中源代码分别作了前向和后向的合并，但是这两个操作逻辑是一样的，因此我们其实可以把他们合并封装在一起，成为一个merge函数，使得逻辑语义更清晰。

代码块

```
//输入: Page *left, Page *right
//处理:
    if (left + left->property == right) {
        left->property += right->property;
```

```
5           ClearPageProperty(right);
6           list_del(&(right->page_link));
7       }
8   //使用：分别执行两次该函数：merge(prev_page, base) 和 merge(base, next_page)
```

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

> 在完成练习一后，参考kern/mm/default_pmm.c对First Fit算法的实现，编程实现Best Fit页面分配算法，算法的时空复杂度不做要求，能通过测试即可。 请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答如下问题：
>
> - 你的 Best-Fit 算法是否有进一步的改进空间?

First Fit /Best Fit 算法对比：

最先匹配法First-fit	最佳匹配法Best-fit
按分区的先后次序，从头查找，找到符合要求的第一个分区就分配。	将分区按小大顺序组织,找到的第一个适应分区是大小与要求相差最小的空闲分区.
分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。	整体来看会形成较多外碎片。但较大的空闲分区可以被保留。

Best Fit 页面分配算法实现：

逻辑关键

Best Fit 关键在于如何找到**大小最合适**的那一块内存

设计思路

对于页面分配来说，我们选择引入一个额外参数min_size，初始化为比空闲页总数还大的值；之后在对空闲页链表进行遍历，记录实时大小“最合适”的页并更新min_size，直到最后遍历结束。而对于页面释放来说，与之前First Fit算法相同，故不再赘述。

代码实现

Best Fit仅仅**只需要修改空闲页选取的逻辑**，其余代码皆和First Fit算法一样，主要修改的地方在下方标黄代码处：

代码块

```
1   //在空闲页链表 free_list 中找到一块大小最合适的连续空闲页，并从中划出n页供使用。
```

```
static struct Page *
Best_Fit_alloc_pages(size_t n) {
    //依然大小检查
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    //遍历空闲链表，寻找大小最合适的空闲页page
    size_t min_size = nr_free + 1; //记录最合适的大小，这里初始化为能表示的最大值
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        // 如果有大小合适的空闲页
        if (p->property >= n) {
            //如果比当前最合适的页还要合适，就选取他
            if (p->property - n < min_size) {
                min_size = p->property - n;
                page = p;
            }
        }
    }
    // 如果找到了合适的page，进行以下操作：
    if (page != NULL) {
        //先取出当前块的前驱结点 prev，再将该空闲块从空闲链表中删除（因为要分配它）；
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        //处理分配后的剩余部分
        if (page->property > n) {
            struct Page *p = page + n;         //新的空闲块头页 = page + n;
            p->property = page->property - n; //它的 property = 原property - n;
            SetPageProperty(p);               //设置 PG_property 位;
            list_add(prev, &(p->page_link));  //把它插在原空闲块的前驱后面，保持地址顺序
        }
        // 减少系统空闲页计数；并清除 PG_property，标记这块页已不再是空闲块;
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

测试结果

将测试目标在pmm.c中改为best_fit算法之后，调用make grade，获取执行结果

```
● ➜ lab2 make grade
  >>>>>>>>>> here_make>>>>>>>>>>>
  gmake[1]: Entering directory '/home/ipv6/OS/lab2' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c + cc kern/driver/
  console.c + cc kern/driver/dtb.c + cc kern/mm/best_fit_pmm.c + cc kern/mm/buddy_system_pmm.c + cc kern/mm/default_pmm.c + cc kern/mm/pmm.c + cc kern/mm/slub_pm
  m.c + cc libs/printfmt.c + cc libs/readline.c + cc libs/sbi.c + cc libs/string.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary b
  in/ucore.img gmake[1]: Leaving directory '/home/ipv6/OS/lab2'
  >>>>>>>>>> here_make>>>>>>>>>>>
  <<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
  try to run qemu
  qemu pid=743025
  <<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
    -check physical_memory_map_information:    OK
    -check_best_fit:                           OK
  Total Score: 25/25
✧₊➜ lab2
```

可以看到，输出测试结果为Best Fit——OK，证明我们的算法编写正确。

算法改进空间

与First Fit算法代码一样，它对于空闲页链表的遍历仍为线性 O(n) 查找，如果空闲页块较多，每次释放或初始化都会线性扫描，性能会急剧下降，故我们还是可以引入一个简单的按地址有序插入二叉搜索树，来把释放或插入复杂度降为 O(log n)。

扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

> 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, 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...
>
> - 参考伙伴分配器的一个极简实现， 在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

buddy system设计文档

1 算法简介

Buddy System 是一种将可用内存划分为大小为 2 的幂次的块来进行管理的分配算法。我们在分配时总是选择能够覆盖请求的最小幂次块，若块比请求大则递归地分裂为“伙伴”块，直到得到合适大小；在释放时与其伙伴检测并尽量合并成更大的块，以减少外部碎片。这套机制用一棵完全二叉树来表达每个节点对应内存区域的最大可用块大小，根节点覆盖整个区域，叶节点对应最小单位页。

本实现参考了wuwenbin的buddy system极简实现，但根据ucore的pmm框架进行了适配和重构，并在一些边界处理和接口兼容方面进行了优化，使其能够无缝集成到ucore操作系统的物理内存管理体系中。在实现顺序上，我们先定义数据结构和树索引宏，然后实现初始化，再完成分配/释放逻辑，最后提供查询与校验。

2 核心数据结构

我们用一个结构体来描述 Buddy System 的核心状态：总页数（2 的幂次）、树数组（每个节点最大可用块大小）、物理页基址。**它借鉴了wuwenbin实现中的二叉树思想，但适配了ucore的Page**

结构。

代码块

```
struct buddy_system {
    unsigned size;          // 管理的总页面数，必须是2的幂
    unsigned *longest;      // 二叉树数组，记录每个节点对应的最大可用块大小
    struct Page *base;      // 内存区域的起始页面
};
```

这里的 `size` 是逻辑管理规模（向上取整到 2 的幂），`longest` 是长度为 `2*size-1` 的完全二叉树数组，`base` 指向 ucore 的 `Page` 数组的起始位置。树的叶子对应最小页，父节点记录左右子树的最大可用块，以便快速定位分配位置。

3 关键宏定义

为了简化代码实现，我们借鉴了wuwenbin实现中的一些关键宏定义，用简单宏来表达树的结构关系与常用判断，便于实现算法时保持代码直观清晰。

代码块

```
#define LEFT_LEAF(index) ((index) * 2 + 1)
#define RIGHT_LEAF(index) ((index) * 2 + 2)
#define PARENT(index) (((index) + 1) / 2 - 1)
#define IS_POWER_OF_2(x) (!((x) & ((x) - 1)))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

`LEFT_LEAF/RIGHT_LEAF/PARENT` 表示在完全二叉树上的索引关系，不需要额外指针；`IS_POWER_OF_2` 用位运算判断幂次；`MAX` 是父节点汇聚左右子状态时的取值规则。

4 pmm_manager结构体

我们在 .c 文件末尾将 buddy System 的各个操作函数以 `pmm_manager` 结构体的方式对外暴露。正如实验课中宫老师所讲，**操作系统设计者用了较为“tricky”的方式，让 C 也能实现类似 C++ 类实例化的效果**：每种新的物理内存分配算法在 .h 中都声明一个 `pmm_manager`，在 .c 末尾都实例化一个对象，像是在实例化一个“类”的对象一样。这使得替换不同算法只需切换这个实例，接口统一、扩展方便。

代码块

```
const struct pmm_manager buddy_system_pmm_manager = {
    .name = "buddy_system_pmm_manager",
    .init = buddy_init,
    .init_memmap = buddy_init_memmap,
    .alloc_pages = buddy_alloc_pages,
```

```
    .free_pages = buddy_free_pages,
    .nr_free_pages = buddy_nr_free_pages,
    .check = buddy_check,
};
```

5 核心函数

接下来，我们展示每个函数关键算法对应的代码片段。**需要说明的是，由于本部分代码量较大，为节省篇幅，不再给出完整代码实现。完整代码及详细注释可参见项目的 GitHub 仓库。**

5.1 buddy_init：初始化状态占位

`buddy_init` 函数负责初始化Buddy System的数据结构。这个函数的设计思路是先进行简单的初始化，实际的详细初始化工作放在 `buddy_init_memmap` 中进行：

代码块

```
static void
buddy_init(void) {
    buddy.size = 0;
    buddy.longest = NULL;
    buddy.base = NULL;
}
```

5.2 buddy_init_memmap：构建树、标记可用叶子、处理非幂次规模

这个函数是整个Buddy System的初始化核心，它根据给定的内存区域初始化Buddy System。我们需要先计算合适的树大小，并初始化所有节点的值。关键算法部分包括：

代码块

```
// size向上取整到2的幂，并限制最大size防止数组越界
unsigned size = 1; while (size < n) size <<= 1;
if (size > 2048) size = 2048;

static unsigned tree[4096];           // 2*2048-1 = 4095
buddy.longest = tree; buddy.size = size; buddy.base = base;
memset(tree, 0, sizeof(tree));

unsigned node_size = size * 2;
for (int i = 0; i < 2 * size - 1; ++i) {
    if (IS_POWER_OF_2(i + 1)) node_size /= 2;
    buddy.longest[i] = node_size;
}

// 屏蔽超出实际页面数的叶子，并自底向上更新父节点
```

```
    if (n < size) {
        for (unsigned i = n; i < size; i++) {
            unsigned index = i + size - 1;
            buddy.longest[index] = 0;
        }
        for (int i = size - 2; i >= 0; i--) {
            unsigned left = buddy.longest[LEFT_LEAF(i)];
            unsigned right = buddy.longest[RIGHT_LEAF(i)];
            buddy.longest[i] = MAX(left, right);
        }
    }
```

这部分代码体现了Buddy System的核心设计思想：将内存划分为2的幂次大小的块，并使用完全二叉树来管理这些内存块。

5.3 buddy_alloc_pages：向下查找合适节点、标记分配、向上汇总

分配指定数量的页面是这个算法的核心功能。我们从根节点开始，递归查找合适大小的空闲块，找到后标记为已分配，并更新父节点的信息。关键算法代码：

代码块

```
// 请求大小向上取整为2的幂
unsigned size = 1; while (size < n) size <<= 1;

// 从根向下选择能容纳size的子树
unsigned index = 0, node_size;
if (buddy.longest[index] < size) return NULL;
for (node_size = buddy.size; node_size != size; node_size /= 2) {
    if (buddy.longest[LEFT_LEAF(index)] >= size) index = LEFT_LEAF(index);
    else index = RIGHT_LEAF(index);
}

// 标记叶子已分配并计算页偏移
buddy.longest[index] = 0;
unsigned offset = (index + 1) * node_size - buddy.size;

// 向上更新父节点的最大可用块大小
while (index) {
    index = PARENT(index);
    buddy.longest[index] = MAX(buddy.longest[LEFT_LEAF(index)],
                               buddy.longest[RIGHT_LEAF(index)]);
}
```

这个算法展示了Buddy System的高效性，通过二叉树结构快速定位到合适的内存块。

5.4 buddy_free_pages：定位节点、标记空闲、伙伴合并

释放指定的页面时，我们需要找到对应的节点，标记为空闲，然后检查伙伴节点是否也空闲，如果是则合并。关键算法：

代码块

```
// 根据页偏移定位叶子索引
unsigned offset = base - buddy.base;
unsigned node_size = 1, index = offset + buddy.size - 1;

// 向上查找“值为0的节点”，找到真实已分配块的叶子
for (; buddy.longest[index]; index = PARENT(index)) {
    node_size *= 2;
    if (index == 0) return; // 防御
}

// 标记空闲并尝试与伙伴合并
buddy.longest[index] = node_size;
while (index) {
    index = PARENT(index); node_size *= 2;
    unsigned L = buddy.longest[LEFT_LEAF(index)];
    unsigned R = buddy.longest[RIGHT_LEAF(index)];
    buddy.longest[index] = (L + R == node_size) ? node_size : MAX(L, R);
}
```

5.5 buddy_nr_free_pages：查询当前最大可分配块

这个函数返回当前可用的页面数，实现非常简单：

代码块

```
static size_t
buddy_nr_free_pages(void) {
    if (buddy.longest == NULL) return 0;
    return buddy.longest[0];
}
```

6 测试用例

测试部分包含两个函数：`basic_check` 和 `buddy_check`。`basic_check` 我们直接复用了 `default_pmm.c` 的实现，而 `buddy_check` 则是我们重点设计的专项测试。

6.1 basic_check

`basic_check` 函数直接复用了 `default_pmm.c` 中的实现，这是为了确保Buddy System能够通过ucore框架的基础测试。这个测试函数主要验证了基本的页面分配和释放功能，包括：

- 分配单个页面并验证分配成功
- 验证分配的页面地址在合理范围内
- 验证页面引用计数正确
- 测试页面释放功能
- 验证内存泄漏检查

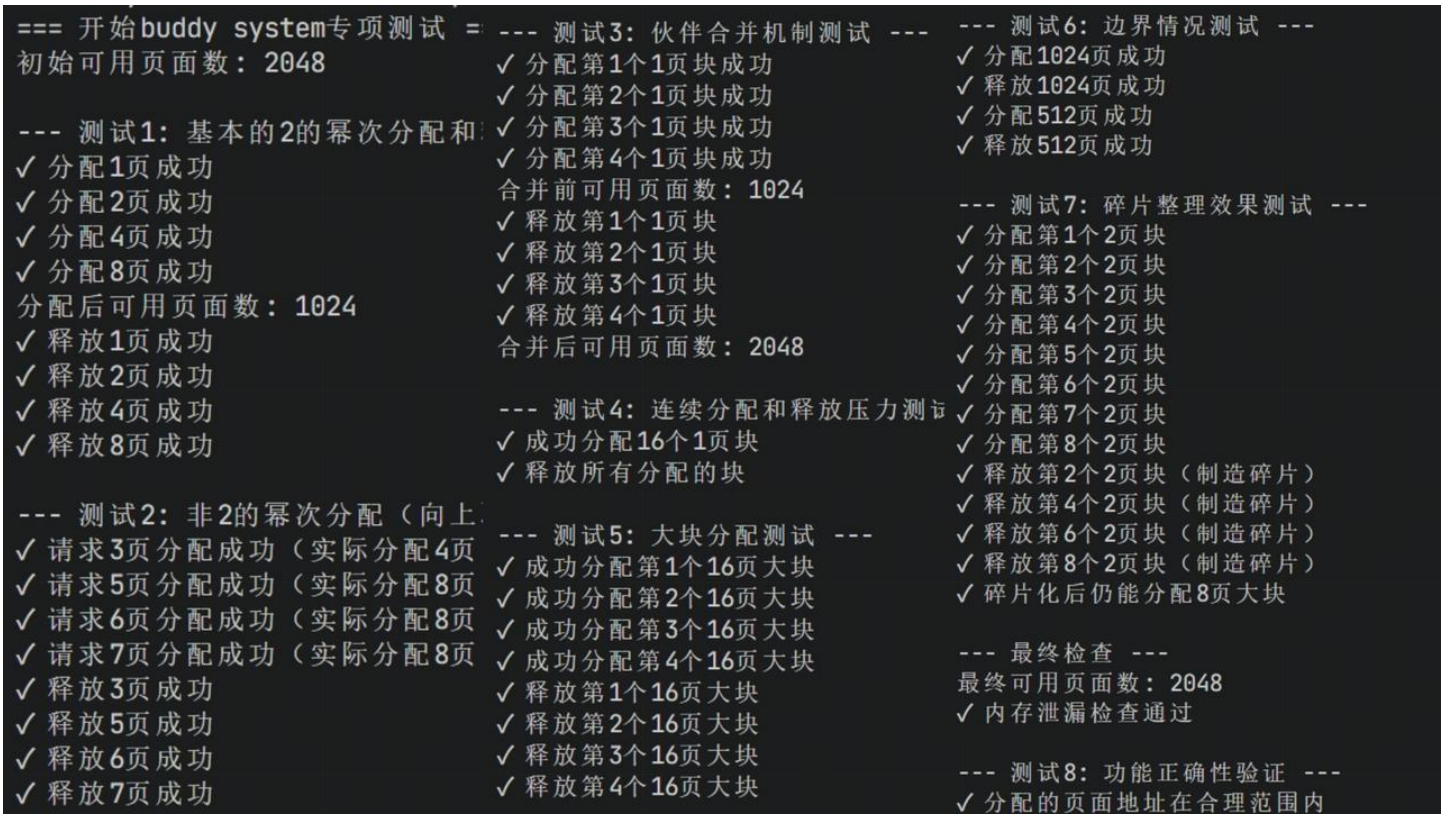
这个基础测试确保了Buddy System与ucore框架的兼容性，为我们后续的专项测试打下了坚实的基础。

6.2 buddy_check函数

`buddy_check` 函数是我们专门为Buddy System设计的综合性测试函数，包含了200多行代码，全面测试了Buddy System的各种功能和边界情况。测试内容主要包括：

- **测试1：基本的2的幂次分配和释放** - 验证Buddy System能够正确分配和释放1、2、4、8页等2的幂次大小的内存块。
- **测试2：非2的幂次分配（向上取整）** - 测试请求3、5、6、7页时，系统能够正确向上取整到4、8、8、8页进行分配。
- **测试3：伙伴合并机制测试** - 验证当连续分配多个小内存块并释放时，系统能够正确合并伙伴块形成更大的空闲块。
- **测试4：连续分配和释放压力测试** - 通过连续分配和释放16个1页块，测试系统在压力情况下的稳定性和性能。
- **测试5：大块分配测试** - 测试分配16页大块的能力，验证系统对大内存请求的处理。
- **测试6：边界情况测试** - 包括测试分配1024页和512页等边界情况，验证系统对极端内存请求的处理。
- **测试7：碎片整理效果测试** - 通过有策略地分配和释放内存块，制造碎片化场景，然后测试系统是否仍能分配大块内存。
- **测试8：功能正确性验证** - 验证分配的页面地址是否在合理范围内，分配的页面地址是否唯一，确保没有地址冲突。
- **测试9：最终内存泄漏检查** - 比较测试前后的可用页面数，确保没有内存泄漏发生。

这些测试用例覆盖全面，输出带有明确的“成功/失败/警告”提示，基本上把 Buddy System 核心行为和边界都跑了一遍，且在 QEMU 下全部通过，确保了算法的正确性和健壮性。最终测试结果展示如下：



由于本部分代码量较大，为节省篇幅，在此不再给出代码实现。

7 总结

总体上，这份设计以“先有数据结构与树宏，才能有初始化；先有初始化，才能保障分配/释放逻辑正确”为主线，算法核心参考了 wuwenbin 的极简实现，但我们在 ucore 的接口规范、边界安全（数组越界与非幂次规模）、测试充分性等方面做了落地适配。接口层通过 `pmm_manager` 的“类实例化”式结构统一暴露，既达到了替换简单、扩展友好的目标，也与课程框架保持一致。

扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

> slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。
>
> - 参考linux的slub分配算法/，在ucore中实现slub分配算法。要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

SLUB设计文档

> **SLAB vs SLUB**：
> 早期 SLAB 通过每个 `kmem_cache` 维护 full/partial/empty 队列与 bufctl/bitmap 记录空闲对象，多核下容易产生锁竞争。SLUB（unqueued） **取消了 SLAB 风格的中心化对象队列与 bufctl，改用**

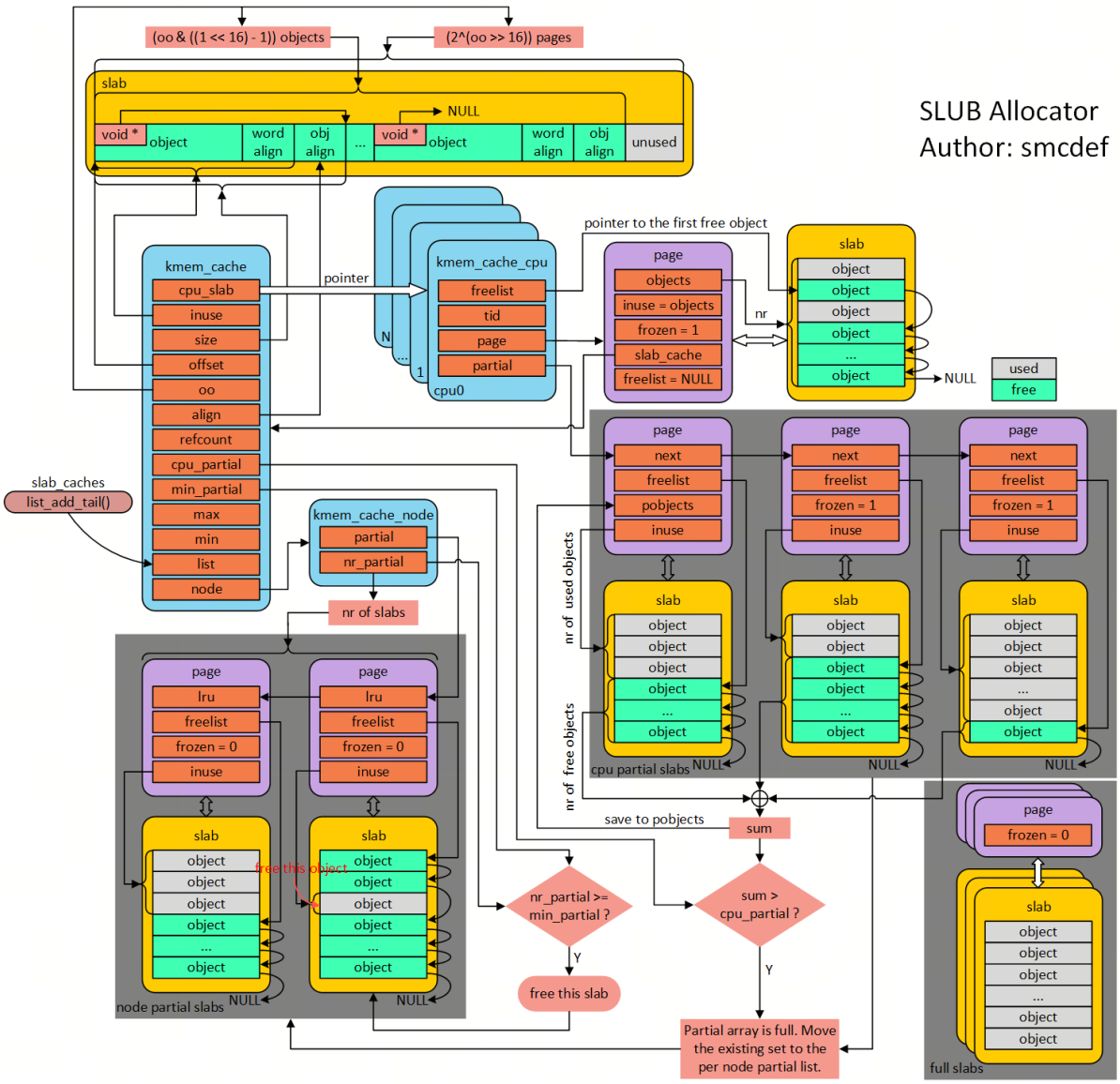
> **对象内嵌 freelist**，并通过每 CPU 当前活动 slab 的快路径降低共享结构争用；慢路径才访问全局/NUMA 级别的 partial 列表，结构更简、扩展性更好。

> **本实验实现的是简化版 SLUB：保留 `kmem_cache`（多种 size）与 `slab`（按页切对象），使用对象内嵌 freelist 实现 O(1) 取还；未实现 Linux 真 SLUB 的 per-CPU 快路径与完整慢路径，但核心思路（两层架构 + 轻量元数据 + 大对象回退页级）已具备。**

> 文中为避免歧义：**“slab”指从 buddy system 拿来的页块（或页组）；“SLUB”指算法/分配器**；代码里 `slub_` 前缀仅表示“本 SLUB 实现的函数或数据结构”。

1 算法简介

SLUB 是一种面向“小对象”的高效内存分配器。它并不直接管理整块物理内存，而是**站在页级分配器（如 Buddy System）肩膀上**工作：先从底层**按页**拿到内存，再把页**切成均匀小对象**，用简洁的空闲链来快速分配与回收。



我的实现是一个简化版 SLUB，重点体现两层架构与核心思路：

- **第一层：页级分配** → 全部委托给已实现的 Buddy System（`alloc_pages` / `free_pages`）

- **第二层：对象级分配** → 在每个 slab（若干页，简化为 1 或 2 页）里按固定对象大小切块，用 `freelist` 管空闲对象；小对象走 SLUB，大对象直接回退到页级。

在 uCore 中，物理内存管理模块通过统一的 `pmm_manager` 接口选择具体实现；我新增了 `slub_pmm_manager` 并在 `pmm.c` 中切换为默认管理器以运行测试。

2 实现思路

2.1 两层架构总览

chatgpt画的好看的图

```
应用/内核对象
      │（按对象大小）
   SLUB 层（对象级）
  ┌──────────────────────────┐
  │ kmem_cache(size=N) │  ← 多个缓存池（32/64/.../2048B）
  │  ├─ slabs_partial  │  ← 还有空位的 slab
  │  └─ slabs_full     │  ← 满的 slab
  └──────────────────────────┘
      │（按页拿/还）
  Buddy System 层（页级）
  ┌──────────────────────────┐
  │ alloc_pages/free...│
  └──────────────────────────┘
```

- **小对象**：落到合适的 `kmem_cache`，从其 `slab` 的 `freelist` 取块；
- **大对象（≥PGSIZE）**：直接走 `alloc_pages`（页级回退），避免 SLUB 头部占用导致放不下的问题。

> 在我的实现里，`2048B` 为了更接近真实 SLUB 的“每 slab 容量更合理”，特地演示了**双页 slab** 的配置，其余 size class 多为单页 slab。

2.2 核心数据结构

> 文中为避免歧义：**“slab”指从 buddy system 拿来的页块（或页组）；“SLUB”指算法/分配器**；代码里 `slub_` 前缀仅表示“本 SLUB 实现的函数或数据结构”。

1. **`kmem_cache`：每种对象大小一个缓存池**

关键信息：`objsize`、`slabs_partial`、`slabs_full`、`slab_pages`（本实验中 1 或 2）。

我在 `slub_init_caches()` 里初始化了一组常见的 size class：`{32, 64, 128, 256, 512, 1024, 2048, 4096}`；并示范把 `2048B` 设置为双页 slab。

代码块

```c
typedef struct kmem_cache {
    const char *name;           // 方便调试的名字
    unsigned int objsize;       // 对象大小
    list_entry_t slabs_partial; // 还有空位的 slab 列表
    list_entry_t slabs_full;    // 已满的 slab 列表
    unsigned int slab_pages;    // 每个 slab 占用的页数（简化为1）
} kmem_cache_t;
```

2. `slub_slab_t`：一个 slab 对应一片连续页

保存这片页的起始 `Page *`、`inuse` 已用计数、`capacity` 总对象数、对象空闲链 `free_list` 等；

代码块

```c
typedef struct slub_slab {
    struct Page *page;          // 这个 slab 对应的起始页
    list_entry_t link;          // 链接到 cache 的 slab 列表
    list_entry_t free_list;     // slab 内部的空闲对象链表
    unsigned int inuse;         // 当前已分配对象数
    unsigned int objsize;       // 对象大小（字节）
    unsigned int capacity;      // 这个 slab 可以容纳的对象总数
} slub_slab_t;
```

简化点：我把 slab 元数据直接放在 slab 页的页头，真实的实现更精细一些，这样方便调试

3. `obj_node`：对象里的小链表头

直接用对象开头的若干字节当 `list_entry_t`，把空闲对象串起来，`O(1)` 取还。

代码块

```c
// 把对象指针挂到 list_entry_t：用对象地址作为 list_entry_t 存储，这里简单把对象首地址当成一个 list 节点
typedef struct obj_node {
    list_entry_t link; // 链表节点在对象头（简化：对象就是裸内存，前几个字节当节点用）
} obj_node_t;
```

2.2 核心函数

2.3.1 slub初始化

- `slub_init()`：初始化全部 `kmem_cache`（size class、链表、`slab_pages`）。
- `slub_init_memmap(base, n)`：**直接把页级初始化转给 Buddy System**，因为 SLUB 不负责底层。

代码块

```
// 初始化所有 size class 的 cache
static void slub_init_caches(void) {
    for (unsigned i = 0; i < SLUB_SIZE_CLASS_COUNT; i++) {
        caches[i].name = "slub";
        caches[i].objsize = slub_sizes[i];
        list_init(&caches[i].slabs_partial);
        list_init(&caches[i].slabs_full);
        caches[i].slab_pages = (slub_sizes[i] == 2048) ? 2 : 1; // 2048B 使用双页 slab
    }
}
```

2.3.2 选 cache：`slub_select_cache(size)`

- 按“**最小可容纳**”原则选择 size class；超出上限则返回 `NULL`，触发页级回退；
- 我保留了 `4096` 这一档用于演示“正好一页”的情况（但由于有 slab 元数据，4096B 实际不走对象级而会回退页级，这里在分配时也做了兜底处理）。

代码块

```
// 根据 size 选择合适的 cache（向上取最近的 size class）
static kmem_cache_t *slub_select_cache(size_t size) {
    for (unsigned i = 0; i < SLUB_SIZE_CLASS_COUNT; i++) {
        if (size <= slub_sizes[i]) return &caches[i];
    }
    return NULL; // 超过 4096B 的对象不适用 SLUB（交给页级）
}
```

2.3.3 创建 slab：`slab_create(cache)`

- **向 Buddy System 申请 `slab_pages` 页**；
- 页头写入 `slub_slab_t` 元数据；
- 计算 `capacity = (total_bytes - sizeof(slab_header)) / objsize`；
- 把每个对象用 `obj_node` 链进 `free_list`。

> 若 `capacity == 0`（比如对象过大），实际系统里要**回退页级**或**拒绝**；我的实现里在对象分配路径已有回退逻辑覆盖。

代码块

```
// 初始化一个 slab，来自 buddy 系统分配的一页
static slub_slab_t *slab_create(kmem_cache_t *cache) {
    struct Page *page = buddy_system_pmm_manager.alloc_pages(cache->slab_pages);
```

```
    if (page == NULL) return NULL;

    // 为 slab 元数据使用这页的起始地址的一小块空间（不再另开页，教学用简化）
    void *page_va = (void *)(page2pa(page) + va_pa_offset);
    memset(page_va, 0, PGSIZE * cache->slab_pages);

    slub_slab_t *slab = (slub_slab_t *)page_va; // slab 元数据放在页头
    slab->page = page;
    list_init(&slab->link);
    list_init(&slab->free_list);
    slab->inuse = 0;
    slab->objsize = cache->objsize;
    unsigned total = PGSIZE * cache->slab_pages;
    unsigned usable = total - sizeof(slub_slab_t);
    slab->capacity = usable / cache->objsize;

    // 切分对象，并把空闲对象都串起来（对象紧随 slab 元数据之后）
    // 注意：这是教学简化，真实 SLUB 会把元数据放在 per-slab 外部或者在对象末尾等，不影响主体思想
    unsigned char *obj_base = (unsigned char *)page_va + sizeof(slub_slab_t);

    for (unsigned i = 0; i < slab->capacity; i++) {
        obj_node_t *node = (obj_node_t *)(obj_base + i * cache->objsize);
        list_add(&slab->free_list, &node->link);
    }

    return slab;
}
```

2.3.4 分配对象：`slub_alloc(size)`

- 先 `slub_select_cache` 找到 cache；若找不到或对象超大，**直接页级回退**（向 Buddy 要整页/多页，返回页的 VA）；
- 否则在 `slabs_partial` 里找第一个能分配出的 slab，走 `slab_alloc_obj` 取一个对象；
- 若 slab 用满了，**把 slab 从 partial 移到 full**；
- 如果 partial 里都塞满了，那就 **`slab_create()`** 一个新 slab，再分配。

代码块

```
// 提供一个对象分配接口（教学扩展），不改变 pmm_manager 的签名，测试中直接调用
static void *slub_alloc(size_t size) {
    kmem_cache_t *cache = slub_select_cache(size);
    if (cache == NULL || cache->objsize == PGSIZE) {
        // 对象太大，直接分配整页（或多页）。这里返回页的虚拟地址（教学简化）。
```

```
        struct Page *pg = buddy_system_pmm_manager.alloc_pages((size + PGSIZE - 1)/PGSIZE);
        if (!pg) return NULL;
        return (void *)(page2pa(pg) + va_pa_offset);
    }

    // 尝试在 partial 列表找到空位对象
    list_entry_t *le = &cache->slabs_partial;
    while ((le = list_next(le)) != &cache->slabs_partial) {
        // 教学简化：把 list 节点本身当做指向 slab 的节点，改为容器获取
        // 我们在创建 slab 时使用 slab->link 作为链入节点，因此这里需要用 to_struct
        slub_slab_t *slab = to_struct(le, slub_slab_t, link);
        void *obj = slab_alloc_obj(slab);
        if (obj != NULL) {
            // 如果分配后满了，移到 full 列表
            if (slab->inuse == slab->capacity) {
                list_del(&slab->link);
                list_add(&cache->slabs_full, &slab->link);
            }
            return obj;
        }
    }

    // 没有可用对象就新建一个 slab
    slub_slab_t *slab = slab_create(cache);
    if (slab == NULL) return NULL;
    // 新 slab 加入 partial 列表
    list_add(&cache->slabs_partial, &slab->link);
    void *obj = slab_alloc_obj(slab);
    if (obj == NULL) return NULL;
    if (slab->inuse == slab->capacity) {
        list_del(&slab->link);
        list_add(&cache->slabs_full, &slab->link);
    }
    return obj;
}
```

2.3.5 释放对象：`slub_free(ptr, size)`

- 通过对象地址**反推所属 slab 的页起始**（因为 slab header 在页头/双页头，很好定位）；
- 把对象挂回 `free_list`，`inuse--`；
- 若刚释放使得 slab 从满变非满，**把 slab 从 full 移回 partial**；
- （**简化取舍**）**暂不做 slab 变空后的“整页回收”**，这样可以减少抖动、便于复用（真实系统会有回收策略，这里留作改进方向）。

```
代码块
1  static void slub_free(void *obj, size_t size) {
2      kmem_cache_t *cache = slub_select_cache(size);
3      if (cache == NULL) {
4          // 当 size>4096 或自定义大对象，调用方需使用 slub_free_pages；这里无法安全定
   位页
5          return;
6      }
7      // 根据对象地址定位所属 slab：对象所在页的开头保存了 slab 元数据
8      uintptr_t obj_pa = (uintptr_t)obj - va_pa_offset;
9      uintptr_t slab_base_pa = ROUNDDOWN(obj_pa, PGSIZE * cache->slab_pages);
10     slub_slab_t *slab = (slub_slab_t *)(slab_base_pa + va_pa_offset);
11     slab_free_obj(slab, obj);
12     // 如果 slab 从 full 变成 partial，需要移动列表
13     if (slab->inuse + 1 == slab->capacity) {
14         // 原来是满的，刚释放后不满
15         list_del(&slab->link);
16         list_add(&cache->slabs_partial, &slab->link);
17     }
18     // 当 slab 变为空，可以考虑回收页：这里教学简化不做回收，保留以降低碎片重新分配成本
19 }
```

2.3.6 页级接口：`slub_alloc_pages / slub_free_pages / slub_nr_free_pages`

- 这三个接口**全部直通 Buddy System**，保持与 `pmm_manager` 其它实现兼容。SLUB 只接管“小对象”的那部分逻辑。

2.3.7 对接 `pmm_manager` & 启动路径

- 新管理器在 `slub_pmm.c` 末尾注册为：`const struct pmm_manager slub_pmm_manager = {... .check = slub_check, ...};`
- `pmm.c` 中选择并打印 `memory management: slub_pmm_manager`；调用 `pmm_manager->init()` / `init_memmap()` / `check()`；
- `pmm.h` 中 `pmm_manager` 接口定义见头文件。

3 测试

3.1 基础函数 `basic_check()`

用 Buddy 连续分配/释放 1 页，确认底层页级 OK；

3.2 自测函数 `slub_check()`

`slub_check()` 是我放在 `slub_pmm.c` 里的自测逻辑，启动时由 `pmm.c` 调用（`pmm_manager->check()`），**只要没 panic 就说明断言全过**，最后会打印 `slub_check()`

`completed.`。

```
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc020201a (virtual)
  edata  0xffffffffc0207018 (virtual)
  end    0xffffffffc020b238 (virtual)
Kernel executable memory footprint: 45KB
memory management: slub_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
slub_check() completed.
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
```

1. **遍历 size class**：
   - `32~2048B`：第一次对象分配应创建 slab，**partial ⩾ 1**；
   - 继续分配到满，slab **从 partial → full**；
   - 再分配一个，**触发第二个 slab**（页地址不同）；
   - 释放一个对象，slab **从 full → partial**；
   - `4096B` 特测：直接按“页级回退”分配/释放（不走对象级）；

代码块

```
// 对象分配测试：覆盖所有 size class，并验证 partial/full 列表移动与跨页扩容
    size_t sizes[] = {32,64,128,256,512,1024,2048,4096};
    for (int si = 0; si < (int)(sizeof(sizes)/sizeof(sizes[0])); si++) {
        kmem_cache_t *cache = slub_select_cache(sizes[si]);
        assert(cache != NULL);
        // 4096B 特例：对象等于一页，走页级路径验证回退逻辑
        if (cache->objsize == PGSIZE) {
            void *ptr = slub_alloc(sizes[si]);
            assert(ptr != NULL);
            uintptr_t page_pa = ROUNDDOWN((uintptr_t)ptr - va_pa_offset, PGSIZE);
            struct Page *pg = pa2page(page_pa);
            buddy_system_pmm_manager.free_pages(pg, 1);
            continue;
        }
        // 新建一个 slab：分配到第一个对象时，partial 应为 1
        void *first = slub_alloc(sizes[si]);
        assert(first != NULL);
        assert(list_count(&cache->slabs_partial) >= 1);

        // 计算单 slab 能装的对象个数（与实现保持一致）
```

```
21          unsigned cap = ((PGSIZE * cache->slab_pages) - sizeof(slub_slab_t)) /
    cache->objsize;
22
23          // 再分配 cap-1 个对象，使该 slab 充满
24          for (unsigned i = 1; i < cap; i++) {
25              assert(slub_alloc(sizes[si]) != NULL);
26          }
27          // 满了之后，partial 至少减少，full 至少增加
28          assert(list_count(&cache->slabs_full) >= 1);
29
30          // 继续分配一个对象，触发第二个 slab 的创建（如果内存允许）
31          void *extra = slub_alloc(sizes[si]);
32          if (extra != NULL) {
33              uintptr_t pa_first = ROUNDDOWN((uintptr_t)first - va_pa_offset,
    PGSIZE);
34              uintptr_t pa_extra = ROUNDDOWN((uintptr_t)extra - va_pa_offset,
    PGSIZE);
35              // 预期来自不同页（不同 slab）
36              assert(pa_first != pa_extra);
37          }
38          // 释放一些对象，观察 partial 列表出现
39          slub_free(first, sizes[si]);
40          assert(list_count(&cache->slabs_partial) >= 1);
41      }
42
```

2. **压力测试（64B × 1024 次）**：交替释放/再分配，验证 freelist 的复用与稳定性；
3. **大对象回退**：`alloc_pages(2)` / `free_pages(2)`，验证与 Buddy 协作；

此外，`pmm.c` 在 `check_alloc_page()` 里还会统一输出 `check_alloc_page() succeeded!`，表示**页级接口**也被系统级测试通过。

扩展练习Challenge：硬件的可用物理内存范围的获取方法（思考题）

> - 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?

方法一：通过固件接口获取内存布局

- 系统上电后，主板固件（如 BIOS 或 UEFI）会在自检POST阶段扫描物理内存，并维护一份精确的内存分布表，记录每个地址段的起始地址、长度和用途类型等。

- 当操作系统或引导程序启动时，可以调用主板固件提供的接口直接读取该内存地图。 OS 只需解析这些条目，筛选出标记为可用的区域即可安全建立自己的物理页管理体系。
- 这种方法完全依赖固件，但缺点是必须在引导阶段调用，且依赖硬件厂商正确实现固件规范。

方法二：由引导加载器传递物理内存映射信息给内核

- 当操作系统本身无法直接访问固件接口，通常由引导加载器在实模式或早期阶段调用固件接口获取内存地图，并在加载内核时将该信息通过启动参数传递给内核。内核启动后直接解析这张表，即可获知哪些物理地址区间可安全使用。
- 这种方法实质上由引导程序完成底层硬件访问，内核只需信任并继承结果。其避免了内核与固件之间的复杂调用。

方法三：通过物理探测机制推测内存边界

- 当系统缺乏固件接口支持、也无引导器提供内存地图，操作系统可通过物理探测方式估算内存可用范围。其核心思路是从一个假定的起始地址开始，按页或大块递增访问物理地址，对每块内存执行读写测试，以判断其是否为真实的可读写内存。
- 若对某区域的写入能被正确读回且不会触发总线错误或机器检查异常，则推断该页属于有效 RAM；反之，则认为是无效或保留区域。该方法能在完全缺乏系统描述表的环境下独立工作，但风险高，错误访问可能引发系统死机或损坏设备。

重要知识点总结

一、实验中的核心知识点及其与OS原理的对应关系

> - 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

- 本实验的重要知识点主要包括页Page结构管理、空闲内存链表的实现、First-Fit内存分配算法、空闲块合并机制等。这些内容在操作系统原理中分别对应于内存管理的分页机制与动态内存分配策略。
- 在操作系统原理中，内存分页是将逻辑内存划分为固定大小的页，并通过页表映射到物理页框，从而实现虚拟地址与物理地址的分离；而在本实验中，`struct Page` 结构体就是对物理页的抽象，每个 Page 对应一页物理内存，是物理内存管理的基本单元；实验中的 free_list 双向链表用于维护所有空闲页块的信息，这与操作系统中空闲区表的概念相对应，只是实现上采用了链表结构以支持动态插入和删除；至于 First-Fit 算法，其思想是“从头开始查找第一个能满足请求的空闲块”。实验中通过 `default_alloc_pages()` 的链表遍历来实现；实验中实现了 `default_free_pages()` 函数，对应 OS 中内存回收与碎片整理的过程，保证了内存释放与块合并。

- 第1个challenge（buddy system）最重要的知识点当然就是buddy system算法，这部分知识点可以直接对应到PPT `4.2OS_MemoryManagement-分页机制-2025` 中的内容。PPT中的内容重理论，学习完其中的内容可以对整个算法有直观的理解。实验中的内容重实践，在用代码实现buddy system的过程中可以加深对算法的理解，关注到一些细节。除此之外，我在实现时参考了wuwenbin的buddy system极简实现，其中的一些巧妙的设计给我留下了比较深刻的印象，这是OS原理课上不会涉及到的。
- 第2个challenge（slub）上课时没有过多提及，所以在下一部分体现。

二、操作系统原理中重要但实验未涉及的知识点

> - 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

1. 虚拟内存管理机制（Virtual Memory Management）

即通过页表（Page Table）、TLB（Translation Lookaside Buffer）等结构实现按需分配与地址空间隔离。这一机制是现代操作系统的核心，但实验中的 physical memory manager 仅在物理层面进行页分配，并未涉及虚拟地址到物理地址的映射。

2. 交换（Swapping）与页面置换算法（Page Replacement）

例如 FIFO、LRU、Clock 等策略，这些算法用于在物理内存不足时将部分页换出到外存。

3. SLUB 内存分配机制

SLUB（**The Unqueued Slab Allocator**）是 Linux 内核中用于高效管理小对象内存的分配器。它在伙伴系统（Buddy System）之上建立了对象级的分配层，将一页或多页内存切分为多个固定大小的对象块，用于频繁创建与销毁的小对象。
其核心思想是：

- **分层设计**：页级内存由 Buddy System 管理，SLUB 负责在页内细粒度地管理对象。
- **缓存复用**：为常见对象大小（如 32B、64B、128B 等）建立对应的缓存（`kmem_cache`），减少重复的初始化和碎片化。
- **快速分配与回收**：通过维护每个缓存的部分使用（`slabs_partial`）、已满（`slabs_full`）等链表，SLUB 能在常数时间内完成对象的分配与释放。

相比传统的 SLAB 算法，SLUB 省去了队列和引用计数等结构，简化了路径，提高了分配速度与并行性能。