Operating System Lab2

禹相祐（2312900）查科言（2312189）董丰瑞（2311973）

> 本次实验我们完成了所有的练习和挑战内容，由查科言负责练习一二和Challenge 3部分，禹相祐负责Challenge 1 Buddy部分，董丰瑞负责Challenge 2 Slub 部分。

一、练习一：理解first-fit 连续物理内存分配算法

> 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?

（1） 算法原理

首次适配算法（first-fit）的原理是将内存中的空闲块按物理地址排列，在进行分配时，**从表头开始扫描**，当找到第一个`size>=n`的空闲分区后，立即停止扫描并进行分配，分配后，如果该空闲分区有剩余空间，将剩余部分作为新的空闲分区保存在链表中。

（2）代码分析

关键数据结构

在解释函数之前，我们需要理解底层依赖的数据结构与宏定义

(1) free_area_t

```
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // number of free pages in this free list
} free_area_t;
```

这个是内存空闲区管理结构体

- `free_list`是双向链表，存储所有空闲物理页块，每个节点是`struct Page`的`page_link`成员。
- `nr_free`是无符号整数，记录系统中空闲物理页的总数量

(2) struct Page

```
 struct Page {
    int ref;                        // 页帧的引用计数器
    uint64_t flags;                 //描述页帧状态的标志位
    unsigned int property;          // 空闲块的页数，用于首次适应算法
    list_entry_t page_link;         //双向链表节点，用于将空闲页组织成链表
};
```

这个是整个内存管理的核心数据结构，每个物理页帧都对应一个Page结构体

- `ref`：引用计数，记录该页被多少个页表项引用
- `flag`：状态标志位，通过bit位表示不同状态
- `property`：当该页是空闲页块的首页时，表示这个空闲块包含的页数
- `page_link`：双向链表结点，用于将空闲页组织成链表

还有两个标志位，定义了页帧的两种关键操作

```
// 标志位定义
#define PG_reserved                 0       // 页是否被内核预留(1=预留,0=可分配)
#define PG_property                 1       // 页是否为空闲块的首页(1=是,0=否)
//操作宏
#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)
```

- **这些宏提供了对页状态操作的便捷方式**

(3) le2page

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

- 这是一个关键的转换宏，通过链表节点（`list_entry_t`)的地址反向计算出包含它的`struct Page`的地址，这是将`Page`结构体组织成链表的基础。

default_init

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

- **作用：初始化空闲页链表和空闲页计数器**，为后续内存管理做准备
  - `list_init(&free_list);` 置空循环链表头
  - `nr_free = 0;` 空闲页计数清零

在代码`libs/list.h`中我们可以找到`list_init`函数的定义

```
static inline void list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;  // 前驱和后继都指向自身，形成循环
}
```

初始化链表头结点时可以调用该函数，`static`确保仅在当前文件可见，`inline`提示编译器将函数代码内联展开，可以减少函数调用的开销。

default_init_memmap

我们在代码里写一些注释便于理解：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);//确保待初始化的页数量大于0，
    struct Page *p = base;//定义一个结构体指针p，指向待初始化区间的起始页
    for (; p != base + n; p ++) {//遍历区间[base,base+n)里的每一页
        assert(PageReserved(p));//这页当前处于“保留/不可分配”状态（早期已标记），进行检查
        p->flags = p->property = 0;//清空flag与property,确保只有块头才会后续设置为property
        set_page_ref(p, 0);//将页面的引用次数设为0
    }
    base->property = n;//把首页作为块头，设置property为总页数n，标识该块的大小
    SetPageProperty(base);// 在块头页上设置 PG_property 标志，标识其为空闲块头
    nr_free += n;//将全局空闲页计数增加n
    if (list_empty(&free_list)) {//判断该链表是否为空
        list_add(&free_list, &(base->page_link));
    } //当前为空，则直接将当前块加入链表
    else {//如果不为空，需要按照物理地址递增的顺序把该块插入到合适的位置
        list_entry_t* le = &free_list;//初始化一个指针指向我们的空闲链表头
        while ((le = list_next(le)) != &free_list) {// 依次遍历所有已存在的空闲块头
            struct Page* page = le2page(le, page_link);//将该链表结点转换为对应的Page结构体，拿到对应的块头Page*
            if (base < page) {//找到第一个地址大于base的块，保证按地址升序
                list_add_before(le, &(base->page_link));//若当前块的起始地址base小于遍历到的页面page的地址，则插入到该
页块之前
                break;//插入完成，退出循环
            } else if (list_next(le) == &free_list) {//如果遍历到最后一个元素仍未插入
                list_add(le, &(base->page_link));//则插入到链表尾部
            }
        }
    }
}
```

- **作用：初始化未被内核占用的物理内存区域**，将一段连续的物理页（从base开始，共n页）标记为空闲，并接入`free_list`，

传入的参数`base`是空闲页块的起始地址（起始页），`n`是页块中包含的物理页数量

首先，我们用断言判定`n`是否大于0，当等于`0`的时候，接入链表没有意义

之后，我们定义一个`Page`类型的指针`p`，指向`base`所指向的内存地址。之后，我们遍历所有的页面，逐页初始化，读取当前对应页面的`PG_reserved`即标志位，判定该页是否是保留页，因为在上电建表前，内核把课管理页先标成不可分配（Reserved），防止被误用，断言确认我们只在这类“尚未纳入空闲链”的页上操作，避免重复初始化或越界。

我们把这些页面变为“空闲”，去掉历史标志，统一置为"空闲的默认状态"，即将`p->flags=0`。将`property`也设为0，后续再仅对块头`base`设置`property=n`，这样只有块头有`PageProperty`，非块头`property=0`。设置`ref=0`，空闲页没有映射和持有者，清理旧址，防止把“曾被用过”的页错误地按在用处理。

在`if`语句中，我们判断该链表是否为空，把新空闲块`base`按物理地址从小到大插入到`free_list`（带哨兵节点的双向循环链表）中

- 如果链表为空，把`base`插到表头后边，相当于第一个元素

- 如果非空，从表头开始按地址递增进行扫描，在找到第一个地址大于新块 `base` 的老块 `page` 时，在 `page` 之前插入 `base`，保证升序，如果已经到了最后一个元素，下一个就是哨兵，则把 `base` 插入到最后一个元素之后，即位于链表尾部。

default_alloc_pages

我们在代码里写一些注释便于理解：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);//请求的页面数量必须大于0
    if (n > nr_free) {
        return NULL;
    }//可用页总数不足，无法分配，返回空
    struct Page *page = NULL;//存储找到的可分配空闲块首页
    list_entry_t *le = &free_list;//从空闲链表头开始遍历
    //循环遍历空闲链表，他的尾节点next指向头，所以终止条件为le回到free_list
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);//利用宏将链表节点转换为对应的Page结构体
        if (p->property >= n) {//检查当前空闲块的大小p->property是否大于请求页数n
            page = p;//找到啦满足条件的块，记录首页
            break;//立即跳出
        }
    }
    if (page != NULL) {
        //记录待删除块的前驱节点
        list_entry_t* prev = list_prev(&(page->page_link));
        //将整个空闲块从空闲链表中删除，因为要分配其中部分页
        list_del(&(page->page_link));
        if (page->property > n) { //如果空闲块数量大于n
            struct Page *p = page + n;//计算剩余块的首页地址：原块首页 + 已分配的n页
            p->property = page->property - n;//设置剩余块的大小：原块大小 - 已分配页数
            SetPageProperty(p); // 标记剩余块为空闲块首页（设置PG_property标志）
            list_add(prev, &(p->page_link));//将剩余块插入到原块的前驱节点之后
        }
        nr_free -= n;//更新全局空闲页总数（减去已分配的n页）
        ClearPageProperty(page);//清除原块首页的“空闲块首页”标志（表示该页已被分配，不再是空闲块
    }
    return page;
}
```

作用：这个函数实现了 `first-fit` 算法的物理页分配。核心功能是从空闲页链表中，找到第一个能容纳 `n` 个连续物理页的空闲块，完成分配并处理块拆分。

这个函数比较好懂，首先我们要进行两个检查，检查请求的页面是否大于 `0`，以及是否有足够的空闲页面可供分配。

之后，从空闲链表的头部开始遍历所有空闲块，找到第一个满足条件即 `p->property>=n` 的块，记录首页。

最后，进行分配，先将整个空闲块从空闲链表中删除，如果空闲块的页面数量大于n，我们需要更新剩余块，将剩余块设为空闲块，插入空闲链表中，将全局的空闲页总数更新减n。返回已分配块的首页，失败了返回 `NULL`

default_free_pages

我们写一些注释便于理解：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);//断言保证释放的页数量大于0
    struct Page *p = base;
    for (; p != base + n; p ++) {//遍历待释放的n个物理页，初始化页的状态
        assert(!PageReserved(p) && !PageProperty(p));// 确保当前页不是预留页也不是空闲块首页
        p->flags = 0;//清除页的所有标志位
        set_page_ref(p, 0); // 将页的引用计数设为0
    }
    base->property = n; // 设置释放块首页的property为总页数n
    SetPageProperty(base);// 标记该页为空闲块的首页
    nr_free += n;// 将释放的n个页加入全局空闲页总数

    //与初始化思路一样，遍历链表，将释放的块插入空闲块链表
    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));
            }
        }
    }
    //尝试与前驱空闲块合并
    list_entry_t* le = list_prev(&(base->page_link));// 获取当前释放块在链表中的前驱节点
     // 若前驱节点不是链表头（即存在前驱空闲块）
    if (le != &free_list) {
        p = le2page(le, page_link);// 将前驱节点转换为Page结构体
          // 若前驱块的末尾地址（p + p->property）等于当前块的起始地址（base），说明地址连续
        if (p + p->property == base) {
            p->property += base->property; // 合并两个块：前驱块的大小 = 原大小 + 当前块大小
            ClearPageProperty(base); // 清除当前块的"空闲块首页"标志（已合并，不再是独立块）
            list_del(&(base->page_link)); // 将当前块从链表中删除（已合并到前驱块）
            base = p; // 更新base指向合并后的块首页（前驱块）
        }
    }
    //尝试与后继空闲块合并，思路和与前驱空闲块合并一致，不再赘述
    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));
        }
    }
}
```

- **作用：该函数用于释放内存块，**完成初始化，将释放的内存块按顺序插入到空闲块链表中，并合并相邻的空闲内存块。

这个函数也比较好懂，首先，我们将释放的页面进行恢复初始化，并且按照地址递增的顺序插入空闲块链表中，这两步与初始化一样，此处不再赘述。

之后，如果相邻的部分，前驱或后继有相邻的空闲块，我们计算地址是否相连`p + p->property == base`，保留块头的`property`，被并入的块头从链表删除且清除`Property`，更新空闲块的页面数量。

default_nr_free_pages

```
static size_t
default_nr_free_pages(void) {
    return nr_free;
}
```

- **作用：获取当前系统中空闲的物理页总数**

basic_check

我们按照理解写一些注释：

```
static void
basic_check(void) {
    // 尝试分配3个单独的物理页，断言分配成功（返回非NULL）
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);
    // 断言3个页的地址互不相同（确保分配的是不同物理页，无重复分配）
    assert(p0 != p1 && p0 != p2 && p1 != p2);
    // 断言3个页的引用计数均为0（新分配的空闲页无引用，符合预期）
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    // 断言3个页的物理地址合法（不超过系统总物理内存大小）
    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);
    // 保存当前空闲链表的状态
```

```
    list_entry_t free_list_store = free_list;
    list_init(&free_list);// 初始化空闲链表为“空链表”（模拟无空闲页的场景）
    assert(list_empty(&free_list));// 断言空闲链表确实为空

    unsigned int nr_free_store = nr_free;
    nr_free = 0;
    // 断言：无空闲页时，分配应失败（返回NULL）
    assert(alloc_page() == NULL);
    // 释放之前分配的3个页
    free_page(p0);
    free_page(p1);
    free_page(p2);
    assert(nr_free == 3);// 断言：释放后空闲页总数应为3（与释放的页数一致）
    // 再次尝试分配3个页，断言分配成功
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);
    // 断言：分配3个页后，无剩余空闲页，再次分配应失败
    assert(alloc_page() == NULL);
    // 释放p0对应的页
    free_page(p0);
    assert(!list_empty(&free_list));// 断言：释放后空闲链表不为空（有1个空闲页，符合预期）

    struct Page *p;
    assert((p = alloc_page()) == p0);// 再次分配1个页，断言分配到的是之前释放的p0（First-fit算法会优先分配第一个空闲页）
    assert(alloc_page() == NULL);// 断言：分配1个页后，空闲页再次为空，分配应失败

    assert(nr_free == 0);// 断言：此时空闲页总数应为0
    free_list = free_list_store;// 恢复空闲链表的原始状态
    nr_free = nr_free_store;// 恢复空闲页总数的原始状态
    //释放测试中分配的最后1个页（p），以及之前的p1、p2，清理测试残留
    free_page(p);
    free_page(p1);
    free_page(p2);
}
```

- **作用：内存管理算法的基础功能检查函数**，通过模拟“分配 - 释放 - 再分配”的简单场景，验证物理页分配（`alloc_page`）、释放（`free_page`）、空闲链表状态、引用计数等核心基础功能是否正常工作，确保内存管理的底层逻辑无错误。

default_check

我们按照理解写一些注释：

```
static void
default_check(void) {
    int count = 0, total = 0;//统计空闲块数量（count）和总空闲页数（total）
    list_entry_t *le = &free_list;
    // 遍历空闲链表，统计空闲块数量（count）和总空闲页数（total）
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p)); // 断言：链表中所有节点都是空闲块首页（PG_property=1）
        count ++, total += p->property; // 累加块数和总页数
    }
    assert(total == nr_free_pages());
//统计的总空闲页数 = 全局空闲页计数（确保链表与nr_free一致，无数据不一致）
    basic_check();//调用basic_check,确保单页分配释放正常

    struct Page *p0 = alloc_pages(5), *p1, *p2;// 分配5个连续物理页，断言分配成功（p0为块首页）
    assert(p0 != NULL);
    assert(!PageProperty(p0));
    // 断言：已分配的块首页p0，不再是空闲块首页（PG_property=0，符合分配后状态）

    list_entry_t free_list_store = free_list;
    list_init(&free_list);// 初始化空闲链表为空（模拟仅释放局部页的场景）
    assert(list_empty(&free_list)); // 断言链表为空
    assert(alloc_page() == NULL);// 断言：空链表时分配失败

    unsigned int nr_free_store = nr_free;
    nr_free = 0;
    //释放p0块中从第3页开始的3个页（p0+2是第3页，共3页）
    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);
    //释放非连续的页（第 1 页和第 3~5 页），验证它们会被标记为两个独立的空闲块，而非错误合并（因中间第 2 页未释放，无法合并）
    p2 = p0 + 1;// p2指向p0块的第2页（未释放）
    free_page(p0);// 释放p0块的第1页（单独1页）
    free_pages(p1, 3);// 释放之前分配的p1块（3页，即原p0+2~p0+4）
    assert(PageProperty(p0) && p0->property == 1);
    assert(PageProperty(p1) && p1->property == 3);
    // 分配1页，断言分配到的是首个空闲块p0（p2-1 = p0，符合First-fit“先找第一个”规则）
    assert((p0 = alloc_page()) == p2 - 1);
    free_page(p0);
    assert((p0 = alloc_pages(2)) == p2 + 1);
    // 测试 “合并空闲块与完整分配”
    free_pages(p0, 2);
    free_page(p2);

    assert((p0 = alloc_pages(5)) != NULL);
    assert(alloc_page() == NULL);
    //恢复系统状态，清理测试环境
    assert(nr_free == 0);
    nr_free = nr_free_store;

    free_list = free_list_store;
    free_pages(p0, 5);

    le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        count --, total -= p->property;
    }
    assert(count == 0);
    assert(total == 0);
}
```

- **作用：这个函数验证`First-fit`内存分配算法的正确性**，通过模拟“多页分配-局部释放-碎片合并”检查算法逻辑正确性，用`assert`断言强制约束每一步的预期结果，确保算法在多页分配、局部释放、碎片合并等关键场景下逻辑无错。

结构体default_pmm_manager

```
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,
};
```

- **作用：这个结构体定义了一个物理内存管理器的实例**，通过结构体`pmm_manager`将内存管理的核心操作（初始化、分配、释放、检查等）封装为统一接口，供操作系统内核调用。
  - `.name = "default_pmm_manager"`：定义管理器的名称
  - `.init = default_init`：函数指针，初始化内存管理器的函数
  - `.init_memmap = default_init_memmap`：函数指针，初始化物理页帧为空闲状态
  - `.alloc_pages = default_alloc_pages`：函数指针，分配连续物理页
  - `.free_pages = default_free_pages`：函数指针，释放物理页
  - `.nr_free_pages = default_nr_free_pages`：函数指针，获取物理页总数
  - `.check = default_check`：函数指针，验证内存管理算法的正确性
- 封装了`First-fit`算法的实现，在我们需要替换为其他算法如`Best-fit`时，只需要定义一个新的`pmm_manager`实例，绑定对应算法的函数，无需修改内核调用内存管理的代码。
- 操作系统启动时，会通过该实例初始化内存管理系统，并在运行过程中通过其提供的函数完成物理页的分配与释放。

（3）总结

经过前边的分析，我们总结一下：

- `default_init`：**初始化空闲页链表和空闲页计数器**，为后续内存管理做准备
- `default_init_memmap(base,n)`：**初始化未被内核占用的物理内存区域**，将一段连续的物理页（从base开始，共n页）标记为空闲，并接入`free_list`
- `default_alloc_pages(n)`：**采用first-fit算法分配内存块**，从空闲页链表中，找到第一个能容纳`n`个连续物理页的空闲块，完成分配并处理块拆分。
- `default_free_pages(base,n)`：**用于释放内存块**，完成初始化，将释放的内存块按顺序插入到空闲块链表中，并合并相邻的空闲内存块
- `default_nr_free_pages()`：**获取当前系统中空闲的物理页总数**
- `basic_check()`：**内存管理算法的基础功能检查函数**
- `default_check()`：**验证`First-fit`内存分配算法的正确性**
- 结构体`default_pmm_manager`：**定义了一个物理内存管理器的实例**

（4）改进空间

我想到了几个方面的改进：

- **减少查找成本：**
  - 采用`Next-fit`，保存上次命中的位置，平均扫描时间更短
  - 按“大小”建索引，同时维护一棵按`property`排序的平衡树/堆。查找 O(log k)，但是需同步维护“按地址”的结构以便合并
  - 采用快速的查找算法，比如二分法
- **减少外部碎片**
  - Best-fit/Good-fit：选择“≥n 的最小块”。
  - 最小剩余阈值：仅当剩余`>= MIN_SPLIT`才拆分，否则整块给出，避免产生碎片屑
- **改进合并策略**
  - 延迟合并：先快速回收到桶，碎片压力或分配失败时再批量合并，提高吞吐。
  - 分配时合并：扫描时遇到相邻小块可临时合并以满足大请求

二、练习二：实现 Best-Fit 连续物理内存分配算法

> 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?

（1）算法原理

`best-fit`最佳适应算法和`first-fit`算法实现类似，核心逻辑是：当有新进程需要内存时，遍历所有空闲分区，**找到大小大于等于进程需求，且与需求差距最小的空闲分区进行分配**，（即找到空闲分区中最小的），若分区有剩余空间，则将剩余部分保留为新的空闲分区。

（2）代码实现

我们的代码中共有5处需要补全，分别位于初始化，分配，释放函数中，只有一处与`first-fit`不一致，即为在`best_fit_alloc_pages`函数中，以下是我补全的代码：

```
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;
    /*LAB2 EXERCISE 2: 2312189*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 遍历空闲链表，查找满足需求的空闲页框
    // 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量

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

```
            if (min_size == n)
                break;
        }
    }
```

可以看到，增加了一个`min_size`变量，用于记录满足要求的最小空闲块的大小，初始值设为总空闲块数量+1，我们遍历所有的空闲块，判断条件是当前块的大小大于等于需求`n`，并且，比我们已经找到的最小块更小即`p->property < min_size`，如果满足条件，更新最优块为当前块，更新最小块大小为当前块的大小。并且，如果我们找到了一个块的大小等于`n`，就可以直接提前`break`，不可能有更好的块了，可以减少循环次数。经过循环后，最小的块即为`page`所指。

（3）结果验证

我们需要更改`pmm.c`中的一行代码，改为调用`best-fit`：

```
static void init_pmm_manager(void) {
    //pmm_manager = &default_pmm_manager;     //first-fit
    pmm_manager = &best_fit_pmm_manager;     // best-fit
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

我们输入`make grade`进行测试，可以得到以下结果：**25/25！所有样例均已通过**

```
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=2759
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check best_fit:                           OK
Total Score: 25/25
```

（4）改进空间

我想了几种改进方向：

- **查找更快：**
  - 分离空闲链表：按块大小分到若干桶（如 1、2、3–4、5–8、…）。在对应桶内做 best-fit
  - 双索引：同时维护“按地址”链表用于合并，和“按大小”的平衡树/跳表/堆用于选最小可用块。
- **碎片更少：**
  - 最小拆分阈值：`if (remain < MIN_SPLIT) 不拆`，整块发放。
  - 舍入到桶：分配请求向上对齐到桶大小（如 2^m），减少尾部碎片
- **混合结构：**
  - 混合算法：可以结合我们之后实现的`buddy`算法，大请求走伙伴（buddy），小请求走 best-fit；兼顾速度、对齐与外碎片。

三、Challenge 1 Buddy 分配器实现

> 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...

（1）Buddy在做什么

按“页”为最小单位分配连续物理内存。块大小总是 **2^o 页**。需要多少页，就找能装下它的最小 2^o 块；不够就向上找更大的，再对半拆。

（2）涉及的核心名词

- **页**：单页大小设置为`2^12B=4KB`。
- **阶 o**：块大小 = `2^o` 页。
- **头页**：块的第一页，带标记`PageProperty=1`表示为头页，带有属性`property=块大小`。
- `free_lists[o]`：每个阶一个空闲链表，只放连续块的**头块**。
- `nr_free`：全局空闲页总数（宏别名`BUDDY_NR_FREE`）。

（3）设计思路

申请 (n) 页：
获取需求 (n) → 分配：向上取整到 (2^k)((n\le 2^k))，若该阶无块则递增 (k) 重试 → 拆分：从拿到的块切出 (n) 页，余块回收至各阶 free list，`buddy_nr_free -= n` → 完成

释放 (n) 页：
释放：用 while 将区间尽量切成“最大且对齐”的 (2^k) 块，`buddy_nr_free += n` → 合并：自小到大找伙伴并合并，无法再变大时挂回对应阶 → 完成

代码实现

函数

- `ORDER_OF_PAGES(o)`：阶数 `o → 2^o` 页。
- `find_upper_number(n)`：将 `n` 向上凑到最小 (2^k)，返回 `k`。
- `turn_page_to_pfn(p)` / `turn_pfn_to_page(pfn)`：`Page* ↔ PFN` 转换。
- `push_block(p, o)`：将以 `p` 为头、大小 (2^o) 的空闲块入 `o` 阶链表，设置头页标记。
- `pop_block(o)`：从 `o` 阶链表取一块，清头页标记并返回头页。
- `remove_block_exact(p, o)`：在 `o` 阶链表精确删除块 `p`。
- `buddy_of(p, o)`：用 `pfn ^ (1<<o)` 求 `p` 的伙伴头页。
- `check_if_head_page(p, s)`：判断 `p` 是否按 `s=2^o` 对齐，是头页则对齐成立。

变量与宏

- `buddy_area.free_lists[o]`：每阶一个双向环链，存空闲头块。
- `buddy_area.nr_free`：当前空闲页总数。
- `BUDDY_NR_FREE`：`buddy_area.nr_free` 的别名。

流程

1) 初始化 logic

- `buddy_init()`：各阶链表设为空环，`BUDDY_NR_FREE = 0`。
- `buddy_init_memmap(base, n)`：将 `[base, base+n)` 切成**对齐的** (2^o) 块，分别插入对应阶链表；保证各阶仅含对齐好的 (2^o) 块。

2) 分块 logic

- `split_block(head, bigger_order, n)`：递归二分为 (2^{o-1}) 与 (2^{o-1})。未用半块回收到对应阶，始终保持两半对齐。

3) 分配 logic

- `buddy_alloc_pages(n)`：计算所需阶 `need`（最小使 (n \le 2^{need})）。自 `need` 向上查找可用块。取到后用 `split` 精确雕刻出 `n` 页；`BUDDY_NR_FREE -= n`。

4) 释放 logic

- `buddy_free_pages(base, n)`：循环将区间切为“最大且对齐”的 (2^o) 块并逐块处理；`BUDDY_NR_FREE += n`。
- `free_one_block_and_merge(p, o)`：从小到大找伙伴，若同阶且空闲则合并升阶；直到无法再合并时挂回对应阶。

5) 校验 logic

- `check_each_list_block(tag)`：统计各阶“块数 × 大小”，应等于 `nr_free_pages()`。
- `buddy_own_check()`：分配 `n=1,2,3`，再乱序释放，核对 `BUDDY_NR_FREE` 的期望变化，符合则输出 `success`。

（4）常用函数作用

- `ORDER_OF_PAGES(o)`：o → 2^o。
- `find_upper_number(n)`：把 n 向上凑到 **2^k**，返回 k。
- `turn_page_to_pfn / turn_pfn_to_page`：Page* 和 物理框页号（PFN） 互转。
- `push_block / pop_block`：头块入表 / 出表。
- `buddy_of(p,o)`：用 `pfn ^ （2^o）` 通过异或的方式寻找buddy块的头。
- `check_if_head_page(p,s)`：p 是否按 s=2^o 对齐。

（5）地址处理：为什么必须“对齐”

- 拆：对半拆分后，拆开的两个一半大小的块理论上都应该是合法头块（只是order减小了1）。
- 合：而在buddy算法中，同阶的伙伴必然紧邻，才能并在一起后成为升了一阶的头块。
- 地址是否对齐的判断方法：`turn_page_to_pfn(p) % (2^o) == 0`。

（6）函数解释

1. 初始化（init_memmap）

清标记 → 计算起始 PFN → 反复选“**最大且对齐**”的 **2^k**：

```
base,n
→ 选 2^k (最大且对齐) → push_block(base,k)
→ base+=2^k, n-=2^k, nr_free+=2^k
→ 循环完毕
```

函数作用的结果：整段的内存会被切成一堆地址对齐好的 ， 大小为 2^k 的头块。

2. 分配（alloc）

如果需要n页大小的内存：

```
需求 n
→ need=find_upper_number(n) → 从阶 need从小到大 找可用块(pop)
→ 递归对半 split：右半回收，左半继续
→ 刚好凑出 n 页 → nr_free -= n
→ 返回这段的头页
```

3. 释放（free）

若要释放 `[base, n)` 这连续n页：

```
while 还有页：
  选“最大且对齐”的 2^k 片 → free_one_block_and_merge(base,k)
  （标为空闲头 → 找同阶伙伴 (pfn与2^o异或) → 能并就摘链合并&升一阶，就像游戏2048）
  base += 2^k
nr_free += n
```

4. 检查正确性(check)

依据

1. 阶 o 的链表里，块都是头页，`property==2^o`，按 2^o 对齐。
2. `nr_free == Σ_o(块数×2^o)`。

检查

- `check_each_list_block`：逐阶统计并检查是否满足：总页数 == `nr_free_pages()`。
- `buddy_own_check`：
  - 分配 1、2、3 共 6 页 → 检查 是否分配了6页出去`-6`；
  - 乱序释放
  - 如果能够回到初值，即从-6变为0, 则成功。

（7）运行结果

> 终端结果

1. 阶段一：BEFORE

```
[BUDDY] before
  order= 0  blk=   1  blocks=    0  pages=      0
  order= 1  blk=   2  blocks=    1  pages=      2
  order= 2  blk=   4  blocks=    0  pages=      0
  order= 3  blk=   8  blocks=    1  pages=      8
  order= 4  blk=  16  blocks=    1  pages=     16
  order= 5  blk=  32  blocks=    1  pages=     32
  order= 6  blk=  64  blocks=    0  pages=      0
  order= 7  blk= 128  blocks=    1  pages=    128
```

```
  order= 8  blk= 256  blocks=    0  pages=      0
  order= 9  blk= 512  blocks=    0  pages=      0
  order=10  blk=1024  blocks=    1  pages=   1024
  order=11  blk=2048  blocks=    1  pages=   2048
  order=12  blk=4096  blocks=    1  pages=   4096
  order=13  blk=8192  blocks=    1  pages=   8192
  order=14  blk=16384  blocks=    1  pages=  16384
  order=15  blk=32768  blocks=    0  pages=      0
  order=16  blk=65536  blocks=    0  pages=      0
  ==> total_pages=31930  nr_free_pages=31930  [YES!]
```

我们可以看到，在初始化后，还未有块内存分配的请求前，依照我们实现的buddy_system的分配逻辑，存在的块的**order**从小到大分别为:1,3,4,5,7,10,11,12,13,14,我们可以算算块大小的总和:

$$\text{总和} = 2^1 + 2^3 + 2^4 + 2^5 + 2^7 + 2^{10} + 2^{11} + 2^{12} + 2^{13} + 2^{14} = 2 + 8 + 16 + 32 + 128 + 1024 + 2048 + 4096 + 8192 + 16384 = 31930$$

我们可以看到——**分配好了的页数 = 总的页数**，说明我们将每个页都成功分配好了。

2. 阶段二：AFTER ALLOC

```
[BUDDY] after alloc
  order= 0  blk=   1  blocks=    2  pages=      2
  order= 1  blk=   2  blocks=    1  pages=      2
  order= 2  blk=   4  blocks=    0  pages=      0
  order= 3  blk=   8  blocks=    0  pages=      0
  order= 4  blk=  16  blocks=    1  pages=     16
  order= 5  blk=  32  blocks=    1  pages=     32
  order= 6  blk=  64  blocks=    0  pages=      0
  order= 7  blk= 128  blocks=    1  pages=    128
  order= 8  blk= 256  blocks=    0  pages=      0
  order= 9  blk= 512  blocks=    0  pages=      0
  order=10  blk=1024  blocks=    1  pages=   1024
  order=11  blk=2048  blocks=    1  pages=   2048
  order=12  blk=4096  blocks=    1  pages=   4096
  order=13  blk=8192  blocks=    1  pages=   8192
  order=14  blk=16384  blocks=    1  pages=  16384
  order=15  blk=32768  blocks=    0  pages=      0
  order=16  blk=65536  blocks=    0  pages=      0
  ==> total_pages=31924  nr_free_pages=31924  [YES!]
[BUDDY] expect change = -6 pages
```

而当我们请求了内存大小n分别为 `1，2，3` 之后，我们看到现在剩下的块，同先前对比一下我们会发现，我们少了一个 `order=3` 即大小 `blk=8` 的块，但我们多了两个 `order=0` 的块，这样实际分配出去的内存的总大小就是

$$(-8) + 2 = -6$$

，符合我们的精准分配的需求。我们也不难想象，大致的分配过程就是:

1. **n=1的请求下：**
   order0 空 → 取 order1 的 2 → 拆成 1+1 → 用走 1，另一块 1 回 order0。
   净变：order1 −1 块，order0 +1 块，页数 −1。
2. **n=2的请求下：**
   order1 空，order2 也空 → 向上取 order3 的 8 → 先拆 8=4+4，把一块 4 回 order2；再把另一块 4 拆 4=2+2 → 用走 2，另一块 2 回 order1。
   净变：order3 −1，order2 +1，order1 +1，页数 −2。
3. **n=3的请求下：**
   此时 order2 有一块 4 → 取这块 4 → 拆 4=2+2：用走一块 2；再把另一块 2 拆 2=1+1：用走 1，另一块 1 回 order0。
   净变：order2 −1，order0 +1，页数 −3。
4. 总结一下:
   order3 −1；order2 净 0；order1 净 0；order0 +2；总页数 −6。
   一共就是——**少了一个 order=3 的 8 页块，多了两个 order=0 的 1 页块**, 我们分析的结果与我们前面的变化一致。

3. 阶段三：AFTER FREE

```
[BUDDY] after free
  order= 0  blk=   1  blocks=    0  pages=      0
  order= 1  blk=   2  blocks=    1  pages=      2
  order= 2  blk=   4  blocks=    0  pages=      0
  order= 3  blk=   8  blocks=    1  pages=      8
  order= 4  blk=  16  blocks=    1  pages=     16
  order= 5  blk=  32  blocks=    1  pages=     32
```

```
  order= 6  blk=  64  blocks=    0  pages=      0
  order= 7  blk= 128  blocks=    1  pages=    128
  order= 8  blk= 256  blocks=    0  pages=      0
  order= 9  blk= 512  blocks=    0  pages=      0
  order=10  blk=1024  blocks=    1  pages=   1024
  order=11  blk=2048  blocks=    1  pages=   2048
  order=12  blk=4096  blocks=    1  pages=   4096
  order=13  blk=8192  blocks=    1  pages=   8192
  order=14  blk=16384  blocks=    1  pages=  16384
  order=15  blk=32768  blocks=    0  pages=      0
  order=16  blk=65536  blocks=    0  pages=      0
  ==> total_pages=31930  nr_free_pages=31930  [YES!]
[BUDDY] expect change = 0 pages
Buddy SUCCESS !
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
```

然后我们再看看乱序释放后的结果，我们释放的顺序分别是: `n = 3 , n = 1, n = 2`。

我们与 alloc 后的结果对比一下，发现：少了两个 `order = 0` 的块，多了一个 `order = 3` 的块，等于回去到了刚初始化后的结果，我们分析下：

1. 释放 n=3：

- 切分：[base,3) = **2 + 1**。
- free 2：作为 **order=1** 入表，找不到 buddy（另一侧 2 在当时被拆成 1+1），暂不合并。
- free 1：与先前遗留的 **1** 合并成 **order=1 的 2**，再与刚刚 free 的 **2** 合并成 **order=2 的 4**；其 buddy 4 不空闲（另一半 4 曾用于分配 n=2 请求的 2），停止。
- 净变化：**order2 +1，order0 −1**，总页 **+3**。

2. 释放 n=1：

- 与它的 buddy **1** 合并成 **order=1 的 2**；其 buddy **2** 不空闲，停止。
- 净变化：**order1 +1，order0 −1**，总页 **+1**。

3. 释放 n=2：

- 与空闲表里的 buddy **2** 合并成 **order=2 的 4**；再与步骤1得到的空闲 **4** 合并成 **order=3 的 8**；更高阶 buddy 不空闲，停止，8 入 **order=3**。
- 净变化：**order3 +1，order2 −1，order1 −1**，总页 **+2**。

汇总：**order3 +1；order2 0；order1 0；order0 −2**（正好吃掉先前新增的两个 1）。总页 **+6**，分布与 **BEFORE** 一致。

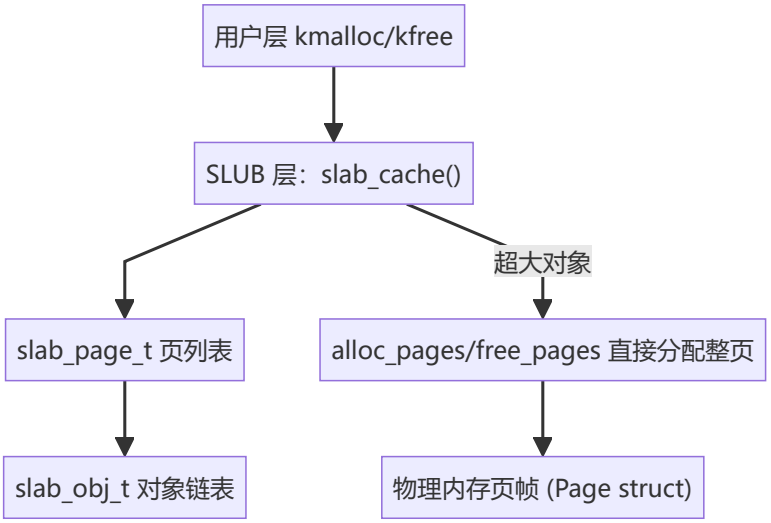
至此证明buddy_system实现正确！

四、Challenge 2 任意大小的内存单元Slub分配算法

> 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。

（1）总体架构

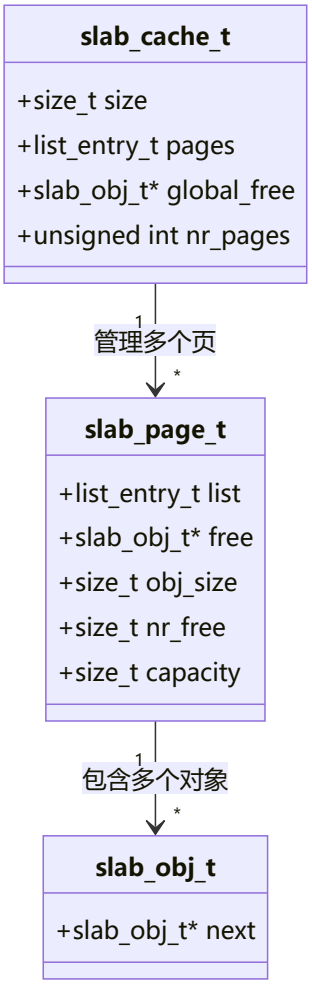
> 体系结构图：



说明：

- 用户通过 `kmalloc` 请求内存；
- SLUB 层根据大小选择对应 `slab_cache`；
- 若页内有空闲对象，从 `slab_page_t.free` 链取出；
- 若无空闲页，向底层 `alloc_pages()` 申请新页；
- `kfree` 则反向释放并可能回收整页。

（2）数据结构关系图



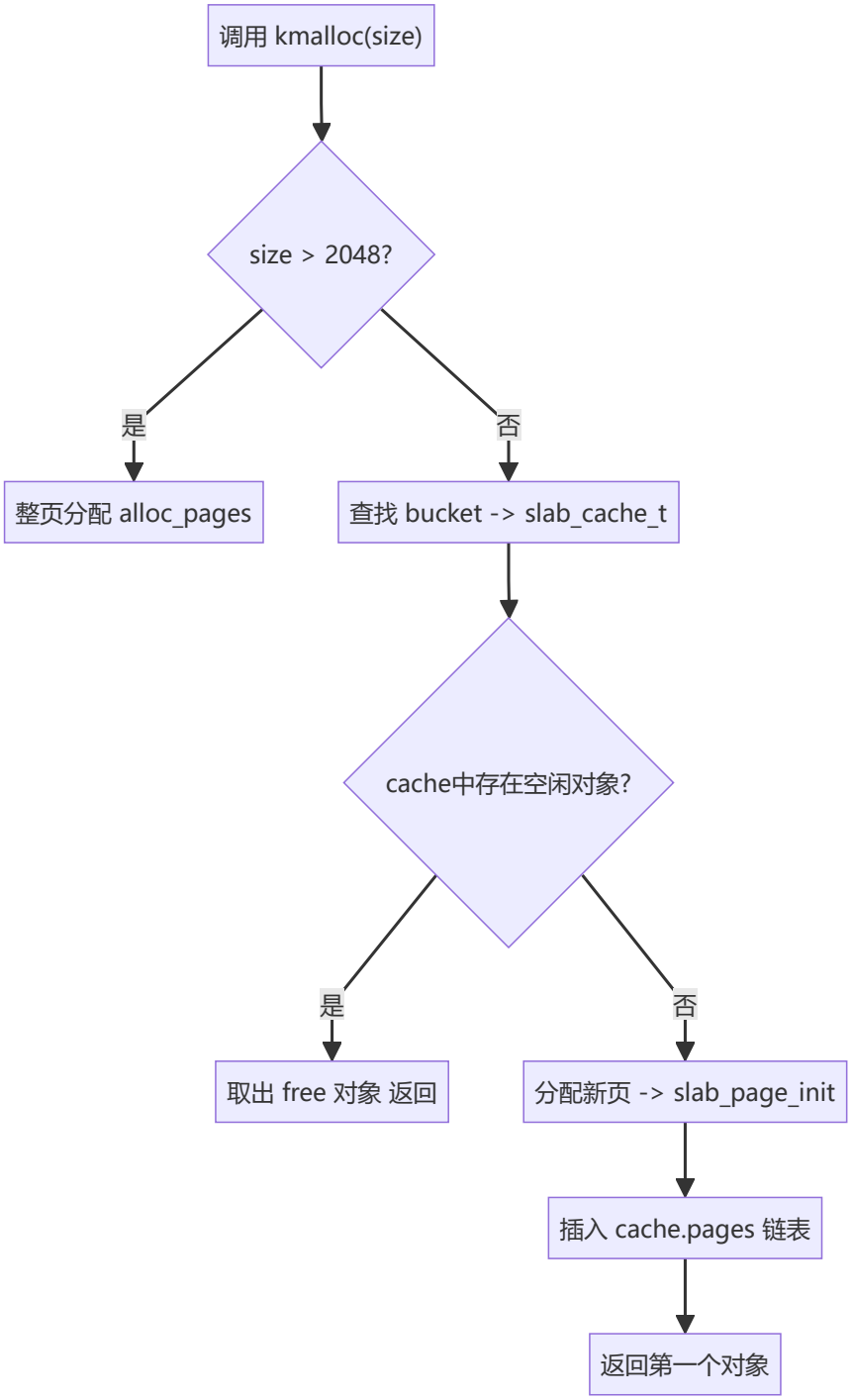
该图展示：

- `slab_cache_t` 管理一个对象大小的所有页；
- 每个 `slab_page_t` 管理一页内的多个 `slab_obj_t`；

- 空闲对象通过单链表链接。

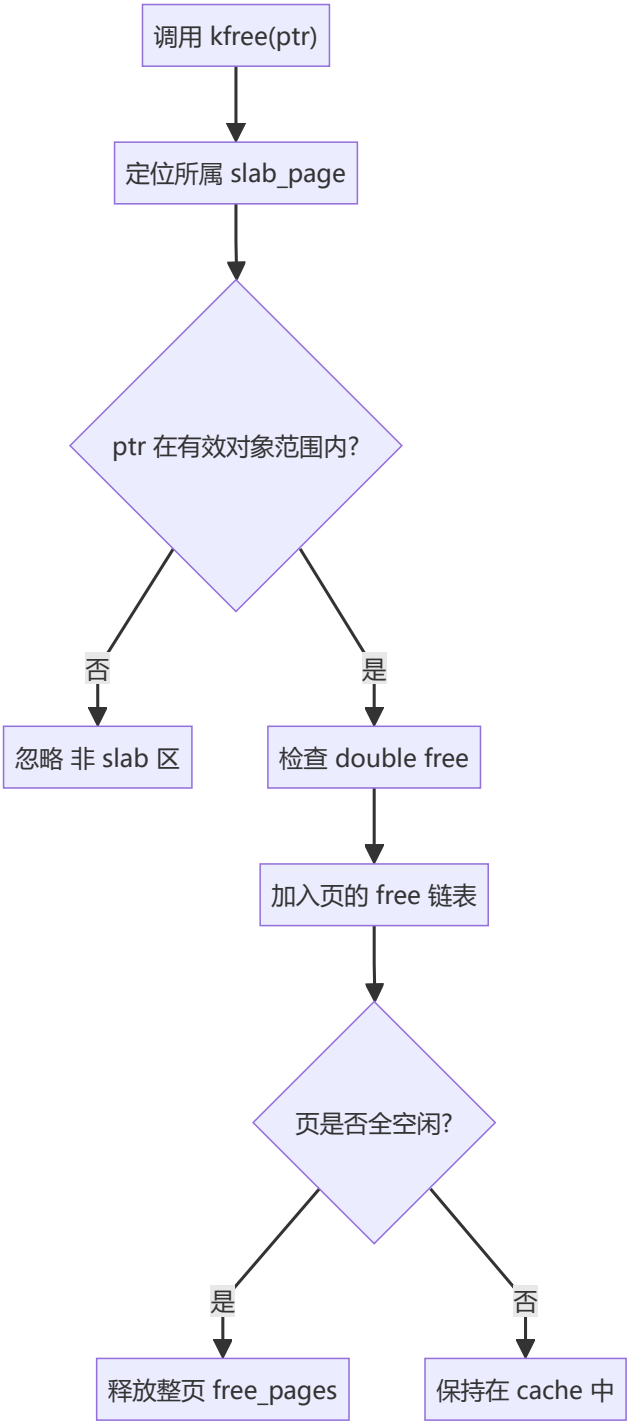
（3）核心函数流程

1. kmalloc 流程图



---

2. kfree 流程图

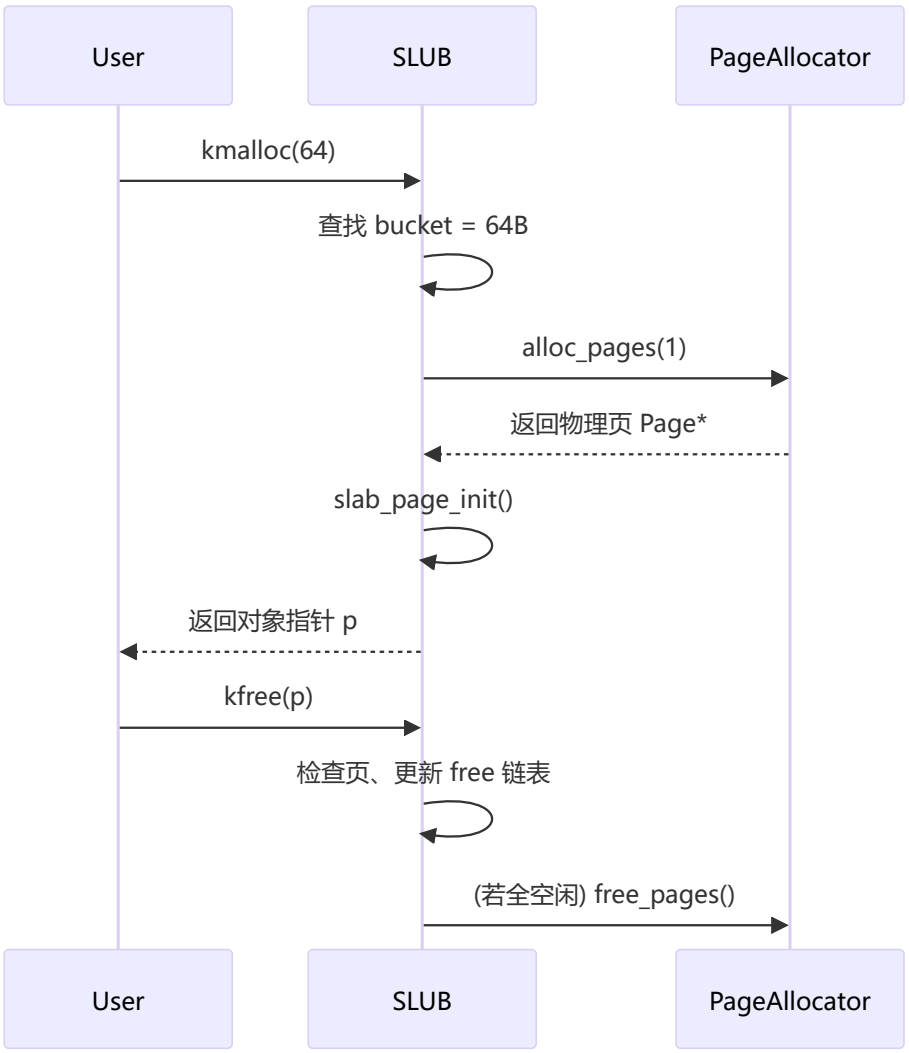


（4）内存布局示意

以 64 字节对象为例（`PAGE_SIZE = 4096`）：

```
+-----------------------------------------------------+
| slab_page_t header (~64B)                           |
+-----------------------------------------------------+
| obj[0]: slab_obj_t + 用户数据 (64B)                  |
| obj[1]: slab_obj_t + 用户数据 (64B)                  |
| obj[2]: slab_obj_t + 用户数据 (64B)                  |
| ...                                                 |
| obj[n]: slab_obj_t + 用户数据 (64B)                  |
+-----------------------------------------------------+
| (空闲对象通过 o->next 链接成单链表)                    |
+-----------------------------------------------------+
```

（5）运行状态图（逻辑时序）



（6）调试输出示例

```
[Buddy] Alloc 1 pages (order 0) @ 0x87fff000
slub 分配 size=8 -> 0xffffffffc7fffff8
[Buddy] Split block: order 3 -> two order 2
[Buddy] Split block: order 2 -> two order 1
[Buddy] Split block: order 1 -> two order 0
[Buddy] Alloc 1 pages (order 0) @ 0x87ff7000
slub 分配 size=24 -> 0xffffffffc7ff7fd0
[Buddy] Alloc 1 pages (order 0) @ 0x87ff8000
slub 分配 size=40 -> 0xffffffffc7ff8fb0
slub 分配 size=56 -> 0xffffffffc7ff8f70
```

SLUB 需要分配一个 8B 对象（kmalloc(8)）

当前 free_area 中没有合适的 order=0 块（1页），buddy 找到一个更大阶的空闲块并经过 split后分配了一个 order=0 块，返回 page 指针，buddy_alloc_pages 打印这行。

SLUB 在得到页后，把页映射成内核虚拟地址 pagev = page2pa(p) + va_pa_offset，然后 slab_page_init() 构造对象链，取出一个对象返回；打印 slub 分配 size=8 -> <virtual>。

在 buddy_init_memmap() 时，free map 是按最大可连块分割放入 free lists（你是从 order = MAX_ORDER-1 往下找最大能放下 n 的 order，然后 list_add）。因此在早期，大多数空闲块可能存在于较高阶（例如 order=9/8/7 等）。

当 buddy_alloc_pages(1) 请求 1 页（order 0）时，若 free_list[0] 为空，buddy_alloc_pages 会向上查找第一个非空阶 cur（这里可能是 3 或更高），取出一个大块，然后逐级分裂（while cur > order），每次分裂都会创建一个右侧 buddy 并插回对应阶的 free_list。

五、Challenge 3 硬件的可用物理内存范围的获取方法

> 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？

（1）手动探测物理内存

- **手动探测物理内存：**经猜测和思考，我们想到可以用“读写验证”的方式主动进行探测，核心是向地址写特定值再读取验证。我们逐块向内存中写入特定测试数据，并读取验证，并且设置异常中断，这样就可以标记区域是否可用。

经过查阅资料，我们对我们设想的方式进行一些补充：

- 探测前的准备工作：
  - **禁用MMU：**确保CPU直接操作物理地址，避免虚拟地址映射干扰结果
  - **确定探测范围：**根据硬件地址总线宽度设定上限，如32位架构最大探测到`0xFFFFFFFF`，64位架构可先探测低40位地址空间，避免超出硬件实际支持范围
  - **选择安全测试值：**使用不会触发硬件异常的特殊值，且避免使用全0或全1，可能与硬件默认状态冲突。
- 分步骤探测
  - **遍历页：**以页面（如 4KB）为单位，从低地址（如 0x00000000）向高地址逐步探测，跳过已知的硬件保留区
  - **读写验证与异常处理**：写入后延迟 1~2 个时钟周期（避免 CPU 缓存导致的 “假读”），再读取该地址的值；若读取值与写入值一致，标记该页为 “临时可用”；若读取时触发硬件异常（如总线错误、非法地址中断），立即标记该区域为 “不可用”（可能是设备寄存器或未安装内存），并跳过后续连续地址（减少异常次数）。
  - **验证后恢复数据**：对 “临时可用” 的页，恢复之前备份的原始数据，避免破坏系统原有状态；合并连续的 “可用页”，形成最终的可用物理内存范围。

（2）读取硬件寄存器

经了解，我们知道，部分硬件会将内存配置信息存储在**内存控制器寄存器**或专用芯片中，操作系统可通过访问这些寄存器直接获取可用内存范围，无需遍历地址。

- **使用内存控制寄存器**

  多数 SoC（系统级芯片）的内存控制器包含 “内存容量配置寄存器”，例如：

- ARM 的 DDR 控制器寄存器（如`DDR_SIZE_REG`）：存储物理内存总容量（如 0x00000001 表示 1GB）；
- RISC-V 的`PMP`（物理内存保护）寄存器：通过配置`PMP`区域，间接确认硬件支持的物理内存范围。

操作系统通过 “内存映射 IO（MMIO）” 方式访问这些寄存器（即通过特定的物理地址读写寄存器值），结合规格书解析得到内存大小和起始地址。

- **使用SPD 芯片**
  - 现代 DDR 内存模组内置**SPD（Serial Presence Detect）芯片**，操作系统通过 I2C 总线。读取 SPD 的 “内存容量” 字段（如第 4~7 字节），计算单条内存的大小；结合硬件支持的内存插槽数量，计算总物理内存容量（如 2 个插槽各插 8GB 内存，总容量 16GB）；再通过内存控制器寄存器确认内存的起始地址，最终确定可用范围。