lab2

李佳：2312668　朱晨瑞：2312674　马溴怡：2311061

练习一 理解 first-fit 连续物理内存分配算法

1.实现过程

First-Fit 算法通过地址升序的双向循环链表管理空闲块，分配时从链表头部开始遍历，找到**第一个**大小满足需求的空闲块即停止，若块过大则**拆分**剩余部分回链表，同时**更新**空闲页计数。

释放时先按地址**升序**将块插入链表，再**合并**前后相邻的空闲块以减少外部碎片，初始化阶段则建立空的空闲链表与计数体系，为后续操作奠定基础。

2.代码分析

(1) default_init

```
static void
default_init(void) {
    list_init(&free_list); // 初始化双向循环链表
    nr_free = 0;           // 初始的无空闲页面
}
```

default_init函数的核心作用是建立物理内存管理的初始空状态。它将全局空闲页面总数nr_free置为0，表明初始状态下没有可分配的物理页面，同时通过list_init(&free_list)初始化空闲链表，这一操作会将链表头节点free_list的prev和next指针均指向自身，形成“自环空链表”。

(2) default_init_memmap

```
static void default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);  // 确保待初始化的页面数不为0
    struct Page *p = base;
    //初始化每一页的基础状态
    for (; p != base + n; p ++) {
        assert(PageReserved(p));  // 确保初始时页面为“保留态”,未被使用
        p->flags = p->property = 0;  // 清除所有标记，非起始页property置0
        set_page_ref(p, 0);        // 将页面引用计数置0，表示无进程引用该页
    }
    //标记“空闲块起始页”
    base->property = n;            // 起始页的property = 空闲块总页数n
    SetPageProperty(base);         // 给起始页打上“空闲块起始”标记
    nr_free += n;                  // 更新全局空闲页面数量
    //按“地址升序”插入空闲链表
    if (list_empty(&free_list)) {
        // 如果空闲链表为空，直接将起始页接入链表
        list_add(&free_list, &(base->page_link));
    } else {
        // 如果链表非空，遍历找到第一个地址>base的节点，在它前面插入
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {  // 遍历链表
            struct Page* page = le2page(le, page_link);  // 链表节点转成Page指针
            if (base < page) {  // 按页面地址比较，base地址更小，插入page前面
```

```
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {  // 遍历到链表尾，插入末尾
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

default_init_memmap的核心逻辑是**将一段连续的物理页面初始化并转化为可分配的空闲块，同时按规则接入空闲链表**，为后续内存分配提供资源。

函数首先通过**assert(n > 0)**进行参数合法性校验，确保待初始化的页面数量不为 0，避免无效操作。随后通过循环遍历base到base + n的所有页面，逐个重置页面状态：

- 先通过assert(PageReserved(p))确保页面初始为 “保留态”（未被其他模块占用，防止重复初始化）
- 再将p->flags（状态标记）和p->property（块大小记录）置 0，调用set_page_ref(p, 0)将页面引用计数清零，让每个页面都处于干净的可分配初始状态。

在单页初始化完成后，函数会对空闲块进行整体标记：

- 将空闲块的起始页base的property设为n，记录整个空闲块的总页数，用于First-Fit 算法中判断块大小
- 通过SetPageProperty(base)为base打上 “空闲块起始页” 的标记，确保后续的链表操作仅针对起始页
- 更新全局空闲页面计数nr_free += n

最后，函数按 “地址升序” 将新空闲块插入free_list：

- 如果链表为空，直接将base的链表节点接入
- 如果链表非空，则遍历链表找到第一个起始地址大于base的空闲块，将base插入它前面，始终保持链表的地址升序特性

在操作系统刚启动时，物理内存大多处于没被管理的原始状态，其中一部分是不能分配的保留页，另一部分是可使用的空闲页。而这个函数会针对这些可使用的页面做两件事：一是把每个页面的状态都初始化好（比如清除无效标记、重置引用计数），二是把这些页面登记到空闲链表中。通过这两步，它能把原本分散的物理页面，整理成有规则的 “空闲块”，最终形成一个能让 default_alloc_pages 函数直接使用的状态。

(3) default_alloc_pages

```
static struct Page *default_alloc_pages(size_t n) {
    assert(n > 0);  // 检查分配页数不为0
    if (n > nr_free) {  // 如果空闲总页数不足，直接返回NULL
        return NULL;
    }
    struct Page *page = NULL;  // 存储找到的空闲块起始页
    list_entry_t *le = &free_list;  // 链表遍历起点

    // 找第一个property≥n的空闲块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);  // 转成Page指针
        if (p->property >= n) {  // 找到“首次满足大小”的块，记录并停止遍历
            page = p;
```

```
            break;
        }
    }

    // 如果找到空闲块，往下处理
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));  // 保存分配块的前驱节点，用于拆分后插回
        list_del(&(page->page_link));  // 将分配块从空闲链表中删除

        // 如果空闲块大小>n，拆分出剩余空闲块并插回链表
        if (page->property > n) {
            struct Page *p = page + n;  // 剩余块的起始页
            p->property = page->property - n;  // 剩余块大小 = 原块大小 - 分配大小
            SetPageProperty(p);  // 标记剩余块为“空闲起始页”
            list_add(prev, &(p->page_link));  // 插入到原分配块的前驱节点后
        }

        // 更新空闲计数，清除分配块的“空闲标记”
        nr_free -= n;  // 空闲页数减少n
        ClearPageProperty(page);  // 分配块不再是空闲块，清除PG_property标记
    }
    return page;  // 返回分配块起始页
}
```

default_alloc_pages 函数是 First-Fit连续物理内存分配算法的核心执行函数，负责将前期通过 default_init_memmap 构建的空闲内存，转化为满足需求的连续物理内存块并分配出去。

函数首先通过assert(n > 0)**确保分配的页数合法。**接着通过全局空闲页计数nr_free进行**快速判断**，如果需求页数n大于nr_free，直接返回NULL，跳过后续链表遍历。之后，函数从空闲链表free_list的表头开始遍历，将链表节点转换为对应的页面指针，逐一检查每个空闲块起始页的property字段，**一旦找到第一个property ≥ n的空闲块，立即停止遍历。**

找到目标空闲块后，函数会先**记录该块的前驱链表节点**，再将其从空闲链表中删除。如果该块的大小大于需求页数n，则会从块的起始页向后偏移n页，得到剩余空闲块的起始页，为剩余块设置property（原块大小减去n）并标记为 “空闲块起始页”，随后**将剩余块插入到原块的前驱节点之后**，确保空闲链表始终保持地址升序的结构。最后，函数更新全局空闲页计数nr_free（减去已分配的n页），并清除已分配块的 “空闲块起始页” 标记。

(4) default_free_pages

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);  // 释放页数不为0
    struct Page *p = base;

    //初始化释放页的状态
    for (; p != base + n; p ++) {
        // 确保释放的页不是保留页、不是空闲块起始页
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;  // 清除所有标记
        set_page_ref(p, 0);  // 引用计数置0
    }

    //标记释放块为“空闲块”，更新空闲计数
```

```
    base->property = n;  // 释放块起始页记录总大小
    SetPageProperty(base);  // 标记为空闲起始页
    nr_free += n;  // 空闲页数增加n

    //按地址升序插入空闲链表（和default_init_memmap的插入逻辑相同）
    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));
            }
        }
    }

    //前向合并
    list_entry_t* le = list_prev(&(base->page_link));  // 释放块的前驱节点
    if (le != &free_list) {  //存在前驱空闲块
        p = le2page(le, page_link);
        // 判断连续，前驱块的起始页+大小=释放块起始页
        if (p + p->property == base) {
            p->property += base->property;  // 合并后大小 = 前驱大小 + 释放块大小
            ClearPageProperty(base);  // 释放块不再是起始页，清除标记
            list_del(&(base->page_link));  // 从链表删除释放块节点
            base = p;  // 合并后的起始页更新为前驱块
        }
    }

    //后向合并
    le = list_next(&(base->page_link));  // 合并后块的后继节点
    if (le != &free_list) {  // 存在后继空闲块）
        p = le2page(le, page_link);
        // 判断连续，合并后块的起始页+大小=后继块起始页
        if (base + base->property == p) {
            base->property += p->property;  // 合并后大小 = 当前大小 + 后继大小
            ClearPageProperty(p);  // 后继块不再是起始页，清除标记
            list_del(&(p->page_link));  // 从链表删除后继块节点
        }
    }
}
```

default_free_pages 函数将进程释放的连续页面**重新纳入空闲区**，并**整理碎片**以提升内存利用率。它先校验释放页面的合法性（非保留页、非重复释放页），重置页面状态与引用计数，确保页面可重新分配；接着将释放的连续页面标记为新空闲块，更新全局空闲页计数，并按地址升序插入空闲链表，维持链表的有序性。

函数最关键的是**双向碎片合并**：分别检查释放块的前、后空闲块，若地址连续则合并为一个大块，清除原小块的空闲标记并删除冗余节点。这一步能有效消除外部碎片，避免小空闲块堆积导致后续大的内存需求无法满足。

(5) basic_check

```
static void
basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    // 测试基础分配功能
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    // 验证分配的页面地址互不相同，避免重复分配同一物理页
    assert(p0 != p1 && p0 != p2 && p1 != p2);
    // 验证新分配页面的引用计数为0，即未被任何进程引用
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    // 验证页面物理地址在有效范围内，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);

    // 保存当前空闲链表和空闲页计数的状态
    list_entry_t free_list_store = free_list;
    list_init(&free_list);  // 重置空闲链表为初始空状态
    assert(list_empty(&free_list));  // 验证链表已清空

    unsigned int nr_free_store = nr_free;
    nr_free = 0;  // 强制设置空闲页计数为0，模拟无内存场景

    // 测试无空闲内存时的分配行为
    assert(alloc_page() == NULL);

    // 释放之前分配的3个页面，验证空闲计数是否正确累加
    free_page(p0);
    free_page(p1);
    free_page(p2);
    assert(nr_free == 3);  // 释放3页后，空闲计数应为3

    // 测试释放后重新分配，应能再次获取3个页面
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    // 验证3页全部分配后，无剩余页面可分配
    assert(alloc_page() == NULL);

    // 测试单一页面释放后的链表状态，链表应非空
    free_page(p0);
    assert(!list_empty(&free_list));

    // 测试释放页面的复用性，再次分配应拿到之前释放的p0
    struct Page *p;
    assert((p = alloc_page()) == p0);
    assert(alloc_page() == NULL);  // 验证分配后链表再次为空
```

```
    // 验证所有页面分配完毕后，空闲计数为0
    assert(nr_free == 0);

    free_list = free_list_store;
    nr_free = nr_free_store;


    free_page(p);
    free_page(p1);
    free_page(p2);
}
```

basic_check通过最小化场景验证内存管理的基础功能，包括单页分配的唯一性、释放后计数是否准确、无内存时的分配失败机制，以及释放页面是否可以复用。它确保了内存分配和释放的最基本逻辑的正确无误。

(6) default_check

```
static void
default_check(void) {
    int count = 0, total = 0;
    list_entry_t *le = &free_list;
    // 遍历空闲链表，验证链表中所有节点均为有效空闲块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p));  // 验证节点是空闲块起始页
        count ++;  // 统计空闲块数量
        total += p->property;  // 累加空闲块总页数
    }
    // 验证空闲块总页数与全局空闲计数一致
    assert(total == nr_free_pages());

    // 先执行基础功能验证
    basic_check();

    // 测试多页分配，分配5个连续页面，验证分配成功且标记正确
    struct Page *p0 = alloc_pages(5), *p1, *p2;
    assert(p0 != NULL);
    assert(!PageProperty(p0));  // 已分配块不应有空闲标记

    // 保存当前状态，创建隔离环境
    list_entry_t free_list_store = free_list;
    list_init(&free_list);  // 清空空闲链表
    assert(list_empty(&free_list));
    assert(alloc_page() == NULL);  // 隔离环境中无内存可分配

    unsigned int nr_free_store = nr_free;
    nr_free = 0;  // 重置空闲计数

    // 释放p0+2开始的3个页面
    free_pages(p0 + 2, 3);
    assert(alloc_pages(4) == NULL);  // 只有3页空闲，无法分配4页
    // 验证释放的块标记正确（是空闲起始页且大小为3）
    assert(PageProperty(p0 + 2) && p0[2].property == 3);
    // 验证重新分配3页时能准确获取释放的块
```

```
    assert((p1 = alloc_pages(3)) != NULL);
    assert(alloc_page() == NULL);  // 分配后无剩余
    assert(p0 + 2 == p1);  // 验证地址正确

    // 测试非连续块释放与分配逻辑
    p2 = p0 + 1;
    free_page(p0);
    free_pages(p1, 3);
    // 验证两个空闲块未合并（地址不连续），属性正确
    assert(PageProperty(p0) && p0->property == 1);
    assert(PageProperty(p1) && p1->property == 3);

    // 验证首次适配算法特性，优先分配低地址块
    assert((p0 = alloc_page()) == p2 - 1);
    free_page(p0);  // 重新释放p0
    // 验证合并后的块能否满足2页分配
    assert((p0 = alloc_pages(2)) == p2 + 1);

    // 释放剩余页面，准备大块分配测试
    free_pages(p0, 2);
    free_page(p2);

    // 验证合并后的块能满足5页分配需求
    assert((p0 = alloc_pages(5)) != NULL);
    assert(alloc_page() == NULL);  // 分配后无剩余

    // 验证隔离环境中空闲计数为0
    assert(nr_free == 0);

    nr_free = nr_free_store;
    free_list = free_list_store;
    free_pages(p0, 5);

    // 验证测试后空闲链表状态与测试前一致
    le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        count --;
        total -= p->property;
    }
    assert(count == 0);  // 空闲块数量恢复初始值
    assert(total == 0);  // 空闲总页数恢复初始值
}
```

作为 First-Fit 算法的验证，它在basic_check的基础之上，重点测试了多页分配、部分释放、碎片合并等复杂情况。通过验证空闲块标记的正确性、链表与计数的一致性、首次适配的地址优先特性，确保算法在边界条件下仍能正确工作。

3.改进空间

（1）优化空闲块查找效率

当前 First-Fit 每次分配需从free_list表头开始线性遍历，若空闲块数量较多，遍历时间会显著增加。

- 现有free_list是单一的地址升序链表，可改为分桶链表。按空闲块大小划分区间，每个区间维护一个地址升序的空闲链表。分配时，先根据需求页数n定位到对应桶，仅遍历该桶的链表，无需遍历全量空闲块，若对应桶无合适块，再向更大的桶查找。
- 现有双向链表仅支持线性遍历，可改用红黑树、AVL 树等平衡二叉树管理空闲块。

（2）减少低地址碎片累积

First-Fit 因优先分配低地址空闲块，容易在低地址区域产生大量无法合并的小碎片，虽然释放时会合并相邻块，但可能仍无法避免碎片累积。

- 可以增加 “主动碎片整理” 功能，当系统空闲时（或碎片率超过阈值），触发整理逻辑，将分散的小空闲块 移动到一起，合并成大块。

练习二 实现 Best-Fit 连续物理内存分配算法

1. 主要思想

最佳适配（Best-Fit） 连续物理内存分配算法的核心原则是：为了有效地减少内存的**外部碎片**，分配器会尽量从所有满足请求的空闲块中，选择**最小的**那个空闲块进行分配。

这样做的好处是，可以保留更大的空闲块来满足未来对大块内存的请求，虽然这可能导致分配速度变慢（因为需要遍历整个空闲列表），但从长远来看，它能更好地管理内存碎片。

2. 实现过程

Best-Fit 算法在分配和回收两个主要方面与 First-Fit 算法有所不同：

操作	Best-Fit 特点
分配 (`alloc_pages`)	**需要遍历整个空闲链表。**算法会记录第一个满足 `n` 个页框请求的空闲块，并继续遍历以寻找一个更小、更"合适"的空闲块。
回收 (`free_pages`)	**必须按页框地址顺序插入。**这与 First-Fit 相同，目的是确保相邻的空闲块可以立即被识别和合并，以最大化连续空闲内存的大小。
分割与合并	分割逻辑与 First-Fit 类似，如果找到的空闲块大于请求大小，则将其分割，剩余部分作为新的空闲块。回收时，必须检查并合并地址相邻的前后空闲块。

3. 代码分析

3.1. `best_fit_alloc_pages(size_t n)`

该函数负责执行最佳适配的查找、分割和分配操作。

```
static struct Page *
best_fit_alloc_pages(size_t n) {
```

```
    // 前置检查
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1; // 初始化为不可能达到的最大值

    // Best-Fit 核心查找逻辑
    while ((le = list_next(le)) != &free_list) {
        struct Page* p = le2page(le, page_link);
        // 遍历链表，查找满足需求且是目前找到的最小块
        if (p->property >= n && p->property < min_size) {
            page = p;
            min_size = p->property;
        }
    }

    if (page) {
        // 如果找到了页，先处理分割
        if (page->property > n) {
            struct Page* new_page = page + n;
            new_page->property = page->property - n;
            SetPageProperty(new_page);
            // 将剩余的空闲部分插入到原页块之后
            list_add(&(page->page_link), &(new_page->page_link));
        }

        // 删除被分配的页块
        list_del(&(page->page_link));

        // 分配后设置标志
        for (struct Page* p = page; p != page + n; p++) {
            ClearPageProperty(p);
            p->flags |= PG_reserved;
        }

        nr_free -= n;
    }
    return page;
}
```

- **Best-Fit 实现点：** 通过循环遍历整个 `free_list`，并使用 `min_size` 变量来追踪并锁定满足条件的最小空闲块，实现了 Best-Fit 的查找策略。
- **分割处理：** 分割后，新的空闲块（`new_page`）被标记属性（`SetPageProperty`）并被重新插入到链表中，紧接着原来的页块。

3.2. `best_fit_free_pages(struct Page *base, size_t n)`

该函数负责回收页块，并按地址顺序插入和执行合并操作。

```
static void
best_fit_free_pages(struct Page *base, size_t n) {
    // 清除页框标志和引用计数

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

```
    nr_free += n;

    // 找到新块的正确插入位置
    list_entry_t* le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page* page = le2page(le, page_link);
        if (base < page) { // 找到第一个地址大于 base 的页块
            break;
        }
    }
    list_add_before(le, &(base->page_link));

    // 尝试与前后相邻的空闲页块进行合并
    struct Page *p = NULL;

    // 向前合并
    list_entry_t* prev_le = list_prev(&(base->page_link));
    if (prev_le != &free_list) {
        p = le2page(prev_le, page_link);
        // 判断前面的页块是否连续：前页块起始地址 + 前页块大小 == 当前页块起始地址
        if (p + p->property == base) {
            p->property += base->property;// 首先更新前一个空闲页块的大小，加上当前页块
的大小
            ClearPageProperty(base);// 清除当前页块的属性标记，表示不再是空闲页块
            list_del(&(base->page_link));// 从链表中删除当前页块
            base = p; // 更新 base 指针，合并后的新块从 p 开始
        }
    }

    // 向后合并
    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        // 判断后面的页块是否连续：当前页块起始地址 + 当前页块大小 == 后页块起始地址
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}
```

- **插入策略：** 通过遍历链表找到按地址升序排列的正确插入位置，确保物理上相邻的块在链表中也是相邻的，这是合并的基础。
- **合并逻辑：** 无论是向前合并还是向后合并，核心都是通过指针算术判断物理地址的连续性。一旦发生合并，就会更新合并后页块的 `property`，清除被合并页块的属性标记，并将其从空闲链表中删除。如果发生向前合并，`base` 指针会更新为前一个块的起始地址，以确保后续的向后合并基于新的合并块。

4.结果

运行 `make qemu`

发现还是使用了`default_pmm_manager`。要测试实现的 Best-Fit 算法，需要在 `kern/mm/pmm.c` 文件中修改 `pmm_manager` 的指向。

下面先运行 `make clean` ,目的是清除之前生成的中间文件防止出现问题。

之后可以运行 `make grade` ，查看评分

测试项	结果	含义
`-check physical_memory_map_information:`	**OK**	内核成功地初始化了物理内存页框，并正确识别、标记和管理了空闲内存块。
`-check_best_fit:`	**OK**	针对 Best-Fit 算法的核心功能测试。这表示代码满足了以下要求： 1. **分配:** 能够遍历空闲列表，找到大小最接近请求页数 N 的空闲块。 2. **分割 :** 当找到的空闲块大于 N 时，能够正确地分割该块，将剩余部分作为一个新的空闲块重新插入链表。 3. **释放与合并:** 能够正确地将释放的页块按地址顺序插入空闲链表，并与前后相邻的空闲块进行合并。
Total Score:	**25/25**	满分，实验完成

扩展练习一： buddy system（伙伴系统）分配算法

设计目标

- 实现符合 “2 的幂次块” 规则的内存管理，所有空闲内存块大小必须为 2^k 页（取值范围 0≤k<MAX_ORDER）。
- 支持高效内存分配，分配时将请求大小向上取整为最近的 2^k 页，若无对应阶数空闲块，则拆分更高阶块直至匹配。
- 支持自动伙伴合并，释放内存时，自动检测相邻伙伴块，合并为更高阶块，直至无法合并。
- 集成到 ucore 现有内存管理框架，通过`pmm_manager`结构体注册算法，与现有 First-Fit/Best-Fit 算法兼容。
- 验证算法的分配正确性、合并有效性及边界场景。

数据结构设计

1.结构体buddy_area_t

```
typedef struct {
    list_entry_t free_list[MAX_ORDER];  // 按阶数划分的空闲块链表数组
    size_t       nr_free;               // 系统总空闲页数
} buddy_area_t;

static buddy_area_t buddy_area;  // 全局伙伴系统管理实例
```

- `free_list[MAX_ORDER]`：数组下标对应阶数（0~MAX_ORDER-1），每个元素是list_entry_t类型的双向链表头，存储对应阶数的空闲块。例如：free_list[4]存储所有大小为 2^4=16页的空闲块。(宏定义MAX_ORDER=15，因 2^{15}=32768 页（每页 4KB 时为 128MB），可覆盖 ucore 实验环境的物理内存规模，避免内存浪费。)
- `nr_free`： 记录系统当前所有空闲块的总页数，用于快速判断分配请求是否可行（若n > nr_free，直接返回 NULL），避免遍历所有链表。

2.辅助工具函数

(1)pages_to_order(size_t n)

- 功能：将 “页数” 转换为对应的 “阶数”（向上取整为最近的 2^k）
- 逻辑：从阶数 0 开始，逐步将tmp=2^{order}与n比较，直至tmp >= n，返回当前order。

```
static inline int
pages_to_order(size_t n)
{
    int order = 0;
    size_t tmp = 1;
    while (tmp < n) {
        tmp <<= 1;
        order++;
    }
    return order;
}
```

(2) order_to_pages(int order)

- 功能：将 “阶数” 转换为对应的 “页数”（即 2^{order}）。
- 逻辑：使用位运算1UL << order

```
static inline size_t
order_to_pages(int order)
{
    return (1UL << order);
}
```

核心功能模块实现

1.初始化模块

(1) buddy_init

- 功能：初始化伙伴系统的全局状态
- 流程：
  - 遍历free_list数组（0~MAX_ORDER-1），调用list_init初始化每个链表头。
  - 将nr_free（总空闲页数）初始化为 0。

```
static void
buddy_init(void)
{
    for (int i = 0; i < MAX_ORDER; ++i)
        list_init(&free_list[i]);  // 初始化所有阶数的空闲链表
    nr_free = 0;                   // 初始无空闲页
}
```

(2) buddy_init_memmap

- 功能：将一段连续的物理内存页（base开始的n页）初始化为伙伴系统的空闲块，并加入对应阶数的空闲链表。
- 流程：
  - 确保n>0且base非空，避免无效操作。
  - 遍历base到base+n的所有页，重置flags（清除保留标志）、property（暂设为 0），并将引用计数ref设为 0。
  - 调用pages_to_order(n)计算n对应的阶数order，得到块大小covered=2^{order}页。若covered > n，则将order减 1，covered更新为$2^{order-1}$)。
  - 设base的property为order，调用SetPageProperty(base)标记其为空闲块起始页。将base的page_link加入free_list[order]链表，更新nr_free。

```
static void
buddy_init_memmap(struct Page *base, size_t n)
{
    assert(n > 0 && base != NULL);
    struct Page *p = base;
    for (; p != base + n; ++p) {
        assert(PageReserved(p));
        p->flags = 0;
        p->property = 0;
        set_page_ref(p, 0);
    }

    int order = pages_to_order(n);
    size_t covered = order_to_pages(order);
    if (covered > n) {                //向上取 2^n 可能越界，回退一级
        order--;
        covered = order_to_pages(order);
    }
    assert(covered <= n);
```

```
    base->property = order;          //记录阶数，而非页数
    SetPageProperty(base);
    list_add(&free_list[order], &(base->page_link));
    nr_free += covered;
}
```

2. 内存分配模块（buddy_alloc_pages）

- 功能：分配n页连续物理内存，返回空闲块的起始页指针，如果分配失败，返回 NULL。
- 流程：
  - 如果n=0或n>nr_free，直接返回 NULL。
  - 调用pages_to_order(n)得到req_order（请求的最小阶数）。
  - 从req_order开始遍历free_list（直至MAX_ORDER-1），找到首个非空的链表（记为order阶），跳至found标签，如果遍历结束未找到，返回 NULL。
  - 如果order > req_order（找到的块阶数高于请求阶数），需要逐级拆分：
    - 从free_list[order]中取出首个空闲块（le = list_next(&free_list[order])），转换为页指针page，并从链表中删除。
    - 计算拆分后的伙伴阶数buddy_order = order - 1，伙伴块大小buddy_pages = 2^{buddy_order}页。
    - 计算伙伴块起始页buddy = page + buddy_pages（地址连续，大小相同）。
    - 标记page和buddy的property为buddy_order，设置PageProperty标志，并将两者加入free_list[buddy_order]链表。
    - 将order更新为buddy_order，重复拆分直至order == req_order。
  - 分配最终块：
    - 从free_list[order]中取出首个空闲块，转换为页指针page，从链表中删除。
    - 清除page的PageProperty标志（标记为已分配），更新nr_free，减去分配的页数。

```
buddy_alloc_pages(size_t n)
{
    if (n == 0 || n > nr_free)
        return NULL;

    int req_order = pages_to_order(n);
    int order;
    for (order = req_order; order < MAX_ORDER; ++order)
        if (!list_empty(&free_list[order]))
            goto found;
    return NULL;                    //没有合适块

found:
    //逐级拆分直到刚好满足
    while (order > req_order) {
        list_entry_t *le = list_next(&free_list[order]);
        struct Page *page = le2page(le, page_link);
        list_del(le);

        int buddy_order = order - 1;
```

```
        size_t buddy_pages = order_to_pages(buddy_order);
        struct Page *buddy = page + buddy_pages;

        page->property = buddy->property = buddy_order;
        SetPageProperty(page);
        SetPageProperty(buddy);

        list_add(&free_list[buddy_order], &(page->page_link));
        list_add(&free_list[buddy_order], &(buddy->page_link));
        order = buddy_order;
    }

    //现在 order == req_order
    list_entry_t *le = list_next(&free_list[order]);
    struct Page *page = le2page(le, page_link);
    list_del(le);
    ClearPageProperty(page);
    nr_free -= order_to_pages(order);

    BS_TRACE("buddy_alloc: %lu pages (order %d) at %p\n",
             order_to_pages(order), order, page);
    return page;
}
```

3.内存释放与合并模块（buddy_free_pages）

- 功能：释放base开始的n页物理内存，并自动合并相邻的伙伴块，加入对应空闲链表。
- 流程：
  - 若n=0，直接返回。确保base非保留页且非空闲块，避免重复释放。
  - 调用pages_to_order(n)得到order，计算covered=2^{order}页。确保covered == n（释放页数必须为 2 的幂，否则为非法操作）。
  - 将base的property设为order，设置PageProperty标志，更新nr_free。
  - 伙伴合并：如果order<MAX_ORDER-1，进入合并循环。
    - 计算base的物理地址base_pa = page2pa(base)，伙伴块物理地址buddy_pa = base_pa (covered * PGSIZE)
    - 将buddy_pa转换为页指针buddy = pa2page(buddy_pa)。
    - 检查伙伴块是否可合并，若!PageProperty(buddy)（伙伴非空闲）或buddy->property != order（伙伴阶数不同），跳出循环，否则继续。
    - 合并操作：
      - 从free_list[order]中删除buddy的链表节点，清除buddy的PageProperty标志。
      - 若base > buddy（确保合并后的块以低地址为起始页），更新base = buddy。
      - 提升块的阶数，order++，设base->property = order，更新PageProperty标志。
  - 将合并后的base加入free_list[order]链表。

```
static void
buddy_free_pages(struct Page *base, size_t n)
{
    if (n == 0) return;
```

```
    assert(!PageReserved(base) && !PageProperty(base));

    int order = pages_to_order(n);
    size_t covered = order_to_pages(order);
    assert(covered == n);

    base->property = order;
    SetPageProperty(base);
    nr_free += covered;

    //向上合并伙伴
    while (order < MAX_ORDER - 1) {
        uintptr_t base_pa  = page2pa(base);
        uintptr_t buddy_pa = base_pa ^ (order_to_pages(order) * PGSIZE);
        struct Page *buddy = pa2page(buddy_pa);

        if (!PageProperty(buddy) || buddy->property != order)
            break;                  //无法合并

        //合并：摘下伙伴，选低地址做头
        list_del(&(buddy->page_link));
        ClearPageProperty(buddy);
        if (base > buddy) base = buddy;

        order++;
        base->property = order;
        SetPageProperty(base);
    }

    list_add(&free_list[order], &(base->page_link));
    BS_TRACE("buddy_free: %lu pages (order %d) at %p\n",
             covered, order, base);
}
```

4.空闲页数统计模块（buddy_nr_free_pages）

- 功能：返回当前系统的总空闲页数。

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

5.自检模块（buddy_check）

- 功能：验证伙伴系统的分配、释放、合并逻辑正确性，输出自检结果。
- 测试用例设计：

编号	名称	目的	步骤
1	最小页分配 / 释放	验证最小单元（1 页，order=0）的分配正确性与释放后空闲页恢复逻辑	1. 记录初始空闲页数nr_orig； 2. 调用alloc_pages(1)分配 1 页； 3. 断言分配成功（p1≠NULL），且空闲页减少 1； 4. 调用free_pages(p1,1)释放； 5. 断言空闲页恢复为nr_orig。
2	非 2 的幂分配 / 释放	验证 “请求页数非 2 的幂时，向上取整为最近 2 的幂” 的分配规则，及释放后合并逻辑	1. 记录当前空闲页数before； 2. 若before<16（10 页向上取整为 16 页），则跳过； 3. 调用alloc_pages(10)分配，断言分配成功； 4. 断言空闲页减少 16（实际分配大小）； 5. 调用free_pages(p1,10)释放； 6. 断言空闲页恢复为before。
3	最大块分配 / 释放	验证系统中 “实际最大空闲块” 的分配与释放正确性，覆盖边界阶数	1. 遍历free_list从最高阶（MAX_ORDER-1）到最低阶，找到首个非空链表，确定max_order（实际最大阶）； 2. 计算最大块页数max_pages=2^{max_order}； 3. 调用alloc_pages(max_pages)分配，断言成功； 4. 断言空闲页减少max_pages； 5. 释放后断言空闲页恢复为nr_orig。
4	多级合并验证	验证 “多块同阶空闲块连续释放时，自动多级合并为高阶块” 的逻辑	1. 记录当前空闲页数before； 2. 若before<32（4×8 页 = 32 页），则跳过； 3. 连续分配 4 个 8 页块（p1~p4）； 4. 连续释放 4 个 8 页块； 5. 断言空闲页恢复为before（验证合并无丢失）。
5	防止超量分配	验证 “请求页数超过总空闲页时，拒绝分配并返回 NULL” 的参数校验逻辑	1. 调用alloc_pages(nr_free_pages()+1)（请求数 = 总空闲页 + 1）； 2. 断言返回值为NULL。
6	重复释放观察	验证 “重复释放已空闲的块时，触发断言防护” 的逻辑（避免非法操作）	1. 分配 1 页块p1； 2. 调用free_pages(p1,1)释放（标记为空闲）； 3. 注释第二次free_pages(p1,1)（避免运行时崩溃），仅说明逻辑。

```
static void
buddy_check(void)
{
    cprintf("======== Buddy System Extended Check Start ========\n");

    struct Page *p1, *p2, *p3, *p4;
    size_t nr_orig = nr_free_pages();   /* 初始空闲页 */
```

```
    /* 1. 最小页（1 页）*/
    cprintf("[TEST 1] 最小页分配（1 页）\n");
    p1 = alloc_pages(1);
    assert(p1 != NULL);
    assert(nr_free_pages() == nr_orig - 1);
    free_pages(p1, 1);
    assert(nr_free_pages() == nr_orig);
    cprintf("        PASS: 1 页分配/释放正确，nr_free 恢复\n");

    /* 2. 非 2 的幂（10 页 → 16 页）*/
    cprintf("[TEST 2] 非 2 的幂分配（10 页，向上取整 16 页）\n");
    size_t before = nr_free_pages();
    if (before < 16) {
        cprintf("        SKIP: 内存不足 16 页\n");
    } else {
        p1 = alloc_pages(10);
        assert(p1 != NULL);
        assert(nr_free_pages() == before - 16);
        free_pages(p1, 10);          /* 按请求数释放 */
        assert(nr_free_pages() == before);
        cprintf("        PASS: 10 页请求实际分配 16 页，释放后合并\n");
    }

    /* 3. 最大块（8192 页）*/
    cprintf("[TEST 3] 边界最大块分配（真实最大阶）\n");
    int max_order = 0;
    for (int i = MAX_ORDER - 1; i >= 0; --i) {
        if (!list_empty(&free_list[i])) {
            max_order = i;
            break;
        }
    }
    size_t max_pages = order_to_pages(max_order);
    if (max_pages == 0) {
        cprintf("        SKIP: 没有空闲块\n");
    } else {
        p1 = alloc_pages(max_pages);
        assert(p1 != NULL);
        assert(nr_free_pages() == nr_orig - max_pages);
        free_pages(p1, max_pages);
        assert(nr_free_pages() == nr_orig);
        cprintf("        PASS: %lu 页（order=%d）分配/释放成功\n",
                max_pages, max_order);
    }

    /* 4. 多级合并（4×8 页 → 32 页）*/
    cprintf("[TEST 4] 多级合并（4×8 页 → 32 页）\n");
    before = nr_free_pages();
    if (before < 32) {
        cprintf("        SKIP: 内存不足 32 页\n");
    } else {
        p1 = alloc_pages(8);
        p2 = alloc_pages(8);
        p3 = alloc_pages(8);
        p4 = alloc_pages(8);
```

```
        free_pages(p1, 8);
        free_pages(p2, 8);
        free_pages(p3, 8);
        free_pages(p4, 8);
        assert(nr_free_pages() == before);
        cprintf("        PASS: 4×8 页 → 1×32 页多级合并成功\n");
    }

    /* 5. 超量分配（nr_free+1）*/
    cprintf("[TEST 5] 超量分配（nr_free+1）\n");
    p1 = alloc_pages(nr_free_pages() + 1);
    assert(p1 == NULL);
    cprintf("        PASS: 返回 NULL，拒绝超量分配\n");

    /* 6. 重复释放（仅观察）*/
    cprintf("[TEST 6] 重复释放防护（仅观察）\n");
    p1 = alloc_pages(1);
    free_pages(p1, 1);
    cprintf("        INFO: 重复释放会触发断言，已跳过第二次释放\n");

    cprintf("======== Buddy System Extended Check Passed ========\n");
}
```

与ucore系统集成

ucore 的内存管理系统通过struct pmm_manager结构体统一管理不同分配算法，需要将伙伴系统注册到该框架。

1.引入头文件

在kern/mm/pmm.c中引入buddy_pmm.h，确保能访问buddy_pmm_manager。

```
#include "buddy_pmm.h"
```

buddy_pmm.h代码：

```
#ifndef KERN_MM_BUDDY_PMM_H
#define KERN_MM_BUDDY_PMM_H

#include "pmm.h"

extern const struct pmm_manager buddy_pmm_manager;

#endif
```

2. 替换默认内存管理器

ucore 使用default_pmm_manager（First-Fit），将其替换为buddy_pmm_manager。

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

测试与结果

我们输入make clean && make -j，如下图所示。

```
ubuntu@Ubuntu-lj:~/Desktop/lab2/lab2$ make clean && make -j
rm -f -r obj bin
+ 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_pmm.c
+ cc kern/mm/default_pmm.c
+ cc kern/mm/pmm.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 bin/ucore.img
```

接着我们输入make qemu来查看测试结果：

```
======== Buddy System Extended Check Start ========
[TEST 1] 最小页分配（1 页）
        PASS：1 页分配/释放正确，nr_free 恢复
[TEST 2] 非 2 的幂分配（10 页，向上取整 16 页）
        PASS：10 页请求实际分配 16 页，释放后合并
[TEST 3] 边界最大块分配（真实最大阶）
        PASS：8192 页（order=13）分配/释放成功
[TEST 4] 多级合并（4×8 页 →32 页）
        PASS：4×8 页 →1×32 页多级合并成功
[TEST 5] 超量分配（nr_free+1）
        PASS：返回 NULL，拒绝超量分配
[TEST 6] 重复释放防护（仅观察）
        INFO：重复释放会触发断言，已跳过第二次释放
======== Buddy System Extended Check Passed ========
```

由此可见，我们的测试用例全部通过。

扩展练习二 任意大小的内存单元slub分配算法

核心思想

slub 的核心思想是通过结构简化与批量优化操作，提升分配效率、降低内存开销并减少碎片。它去掉了 SLAB 中冗余的缓存链等中间层级，每个内存缓存直接关联存储对象的 slab 页，大幅减少元数据开销，同时 slab 页内直接存储待分配的对象，只通过页描述符记录空闲数量等关键状态，进一步提升内存利用率。SLUB还采用 per-CPU 缓存机制减少锁竞争，分配与回收时优先操作本地缓存，通过批量迁移对象降低频繁操作成本，同时动态调整 slab 页大小适配不同需求，有效抑制内存碎片产生。

设计目标

- 实现**两层架构**的内存分配机制，第一层基于物理页进行内存管理，提供页级别的分配与释放，第二层在页级基础上实现任意大小的内存对象分配，支持不同尺寸的高效管理。
- 采用**缓存机制**，为常用大小的对象（预定义 32B、64B、128B）维护专用缓存池，减少内存碎片并提升分配效率。

- 每个缓存池由多个slab组成，每个slab对应一个物理页，内部划分多个相同大小的对象，通过位图（bitmap）跟踪对象的分配状态。
- 支持自动slab管理，当缓存池无空闲对象时自动创建新slab，当slab中所有对象均被释放时，自动回收对应的物理页。
- 集成到现有内存管理框架，通过pmm_manager结构体注册算法，兼容系统的初始化、内存映射及测试流程。
- 验证算法的分配正确性、释放安全性及边界场景处理能力。

数据结构设计

1.核心结构体定义

(1) slab_t：管理单页内的对象集合

```
typedef struct Slab {
    list_entry_t list;        // 用于链接到所属Cache的链表节点
    size_t free_cnt;          // 空闲对象数量
    void *objs;               // 对象存储区起始地址（内核虚拟地址）
    unsigned char *bitmap;    // 位图，标记对象分配状态（1=已分配，0=空闲）
} slab_t;
```

(2) cache_t：管理相同大小对象的缓存池

```
typedef struct Cache {
    list_entry_t slabs;       // 管理的Slab链表（包含所有状态的Slab）
    size_t obj_size;          // 单个对象的大小（固定值，如32B）
    size_t objs_num;          // 每个Slab中可容纳的对象数量
} cache_t;
```

(3) 页级内存管理结构

```
static free_area_t free_area;
#define free_list (free_area.free_list)  // 空闲页链表（地址升序）
#define nr_free (free_area.nr_free)      // 总空闲页数
```

2.辅助工具函数

(1) calculate_objs_num(size_t obj_size)

- 功能：计算单个slab（1 页，4KB）可容纳的对象数量。
- 逻辑：总可用空间为`PGSIZE - sizeof(slab_t)`，每个对象需`obj_size`字节，位图每 8 个对象需 1 字节，因此计算公式为：

```
(PGSIZE - sizeof(slab_t)) / (obj_size + 1.0/8.0)
```

(2) KADDR（pa）

- 功能：将物理地址转换为内核虚拟地址。
- 逻辑：根据内存布局，内核虚拟地址 = 物理地址 + PHYSICAL_MEMORY_OFFSET。

```
#define KADDR(pa) ((void *)((uintptr_t)(pa) + PHYSICAL_MEMORY_OFFSET))
```

核心功能模块实现

1.初始化模块

(1) slub_init(void)

- 功能：初始化 SLUB 分配器的两层架构。
- 流程：
  - 调用default_init()初始化第一层（页级分配器），创建空的free_list并置nr_free为 0。
  - 调用 `cache_init()` 初始化第二层（对象缓存），创建 3 种预定义大小的 `cache_t` 结构。

```
static void slub_init(void) {
    default_init();          // 初始化第一层：页级分配器
    cache_init();            // 初始化第二层：SLUB缓存
}
```

```
static void default_init(void) {
    list_init(&free_list);
    nr_free = 0;  // 初始无空闲页
}
```

```
static void cache_init(void) {
    cache_n = 3;
    size_t sizes[3] = {32, 64, 128};  // 三种对象大小

    for (int i = 0; i < cache_n; i++) {
        caches[i].obj_size = sizes[i];
        caches[i].objs_num = calculate_objs_num(sizes[i]);
        list_init(&caches[i].slabs);// 初始化slabs链表为自环空列表
    }
}
```

(2) slub_init_memmap(struct Page *base, size_t n)

- 功能：将物理页范围初始化为可管理的空闲页。
- 流程：复用页级初始化函数default_init_memmap，将base开始的n页标记为空闲，按地址升序插入free_list，更新nr_free。

```
static void slub_init_memmap(struct Page *base, size_t n) {
    default_init_memmap(base, n);
}
```

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

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

```
    base->property = n;  // 标记为空闲块头部
    SetPageProperty(base);
    nr_free += n;

    // 将新的空闲块插入到合适位置，保持链表升序
    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));
            }
        }
    }
}
```

2. 页级分配与释放模块

(1) default_alloc_pages(size_t n)

- 功能：从free_list分配n个连续物理页（第一层分配）。
- 流程：采用 First-Fit 算法，遍历free_list找到第一个大小≥n的空闲块，拆分后返回起始页，更新nr_free。

```
static struct Page *default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;  // 内存不足
    }

    struct Page *page = NULL;
    list_entry_t *le = &free_list;

    // 查找第一个足够大的空闲块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }

    if (page != NULL) {
        list_entry_t *prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));

        // 如果有剩余空间，分割成新的空闲块
        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;
}
```

(2) default_free_pages(struct Page *base, size_t n)

- 功能：释放base开始的n个物理页（第一层释放）。
- 流程：重置页状态，按地址升序插入free_list，并与前后相邻空闲块合并以减少碎片，更新nr_free。

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

    // 重置页属性
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }

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

    // 将释放的页插入到合适位置
    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));
            }
        }
    }

    // 与前一个空闲块合并
    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;
        }
    }

    // 与后一个空闲块合并
    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));
        }
    }
}
```

3. 对象级分配与释放模块

(1) slub_alloc_obj(size_t size)

- 功能：分配大小为size的对象（第二层分配）。
- 流程：
  - 查找合适的缓存池，选择obj_size≥size的最小cache_t 。
  - 遍历缓存池的slabs链表，寻找有空闲对象的slab。如果找到，通过位图查找第一个空闲对象，free_cnt减 1，返回对象地址。
  - 如果无可用slab，调用create_slab创建新Slab：
    - 分配 1 个物理页，转换为内核虚拟地址，初始化slab_t结构。
    - 将新Slab加入缓存池的slabs链表。
  - 从新slab分配第一个对象，返回其地址。

```
static void *slub_alloc_obj(size_t size) {
    if (size <= 0) {
        return NULL;
    }

    // 查找合适的缓存
    cache_t *cache = NULL;
    for (int i = 0; i < cache_n; i++) {
        if (caches[i].obj_size >= size) {
            cache = &caches[i];
            break;
        }
    }
    if (cache == NULL) {
        return NULL;  // 没有合适的缓存
    }

    // 在缓存的Slab链表中查找有空闲对象的Slab
    list_entry_t *le = &cache->slabs;
    while ((le = list_next(le)) != &cache->slabs) {
```

```
        slab_t *slab = le2slab(le, list);
        if (slab->free_cnt > 0) {
            // 查找第一个空闲对象
            for (size_t i = 0; i < cache->objs_num; i++) {
                size_t byte = i / 8;
                size_t bit = i % 8;
                if (!(slab->bitmap[byte] & (1 << bit))) {
                    // 标记为已分配
                    slab->bitmap[byte] |= (1 << bit);
                    slab->free_cnt--;
                    // 返回对象地址
                    return (void *)slab->objs + i * cache->obj_size;
                }
            }
        }
    }

    // 没有可用的Slab，创建新的Slab
    slab_t *new_slab = create_slab(cache->obj_size, cache->objs_num);
    if (!new_slab) {
        return NULL;  // 内存不足
    }

    // 将新Slab加入缓存的Slab链表
    list_add(&cache->slabs, &new_slab->list);
    // 分配第一个对象
    new_slab->bitmap[0] |= 1;  // 标记第一个对象为已分配
    new_slab->free_cnt--;

    return new_slab->objs;
}
```

(2) create_slab(size_t obj_size, size_t objs_num)

- 功能：创建新的slab。
- 流程：
  - 调用default_alloc_pages(1)分配 1 个物理页。
  - 将物理页地址转换为内核虚拟地址，作为slab_t结构体的起始地址。
  - 初始化slab_t成员。
  - 返回slab_t指针。

```
static slab_t *create_slab(size_t obj_size, size_t objs_num) {
    // 分配一个物理页
    struct Page *page = default_alloc_pages(1);
    if (!page) {
        return NULL;
    }

    // 转换为内核虚拟地址
    void *kva = KADDR(page2pa(page));
    slab_t *slab = (slab_t *)kva;

    // 初始化Slab结构
```

```
    slab->free_cnt = objs_num;
    slab->objs = (void *)slab + sizeof(slab_t);  // 对象存储区起始地址
    // 位图起始地址 = 对象存储区起始地址 + 对象总大小
    slab->bitmap = (unsigned char *)((void *)slab->objs + obj_size * objs_num);
    // 初始化位图(全部置0，表示所有对象空闲)
    memset(slab->bitmap, 0, (objs_num + 7) / 8);
    list_init(&slab->list);

    return slab;
}
```

(3) slub_free_obj(void *obj)

- 功能：释放对象（第二层释放）。
- 流程：
  - 遍历所有缓存池和Slab，定位obj所属的Slab和缓存池。
  - 计算obj在slab中的索引，在 bitmap 中标记为空闲，free_cnt加 1。
  - 清空对象内存。
  - 若slab的free_cnt等于objs_num（所有对象空闲）：
    - 将slab从缓存池链表中移除。
    - 调用default_free_pages释放slab对应的物理页，回收至页级空闲链表。

```
static void slub_free_obj(void *obj) {
    // 查找对象所属的slab和cache
    for (size_t i = 0; i < cache_n; i++) {
        cache_t *cache = &caches[i];
        list_entry_t *le = &cache->slabs;

        while ((le = list_next(le)) != &cache->slabs) {
            slab_t *slab = le2slab(le, list);
            // 检查对象是否在当前slab的对象存储区内
            if (obj >= slab->objs && obj < (slab->objs + cache->obj_size * cache-
>objs_num)) {
                // 计算对象在slab中的索引
                size_t offset = (char *)obj - (char *)slab->objs;
                size_t index = offset / cache->obj_size;
                size_t byte = index / 8;
                size_t bit = index % 8;

                // 标记为未分配
                if (slab->bitmap[byte] & (1 << bit)) {
                    slab->bitmap[byte] &= ~(1 << bit);
                    slab->free_cnt++;
                    // 清空对象内存，避免信息泄露
                    memset(obj, 0, cache->obj_size);

                    // 如果slab中所有对象都已释放，回收整个slab
                    if (slab->free_cnt == cache->objs_num) {
                        list_del(&slab->list);
                        // 释放slab对应的物理页
                        struct Page *page = pa2page(PADDR(slab));
                        default_free_pages(page, 1);
```

```
                    }
                }
                return;
            }
        }
    }
}
```

4.空闲页统计（slub_nr_free_pages(void)）

返回系统当前空闲页总数，直接返回nr_free。

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

5. 测试模块（slub_check(void)）

- 功能：验证 SLUB 算法的正确性。
- 测试用例设计：

编号	测试内容	目的	关键步骤
1	边界检查	验证无效输入处理	分配 0 字节或超过最大缓存（256B）的对象，断言返回 NULL。
2	基本分配 / 释放	验证单对象分配释放逻辑	分配 32B 对象，写入数据后验证；释放后重新分配，验证内存已清零。
3	多对象管理	验证批量分配的正确性	分配 10 个 64B 对象，写入不同数据后验证；释放后验证内存清零。
4	混合分配 / 释放	验证缓存池与 Slab 的动态管理	混合分配 32B、64B、128B 对象，验证页数量变化；释放后检查是否回到初始状态，验证 Slab 回收逻辑。

```
static void slub_check(void) {
    cprintf("Starting SLUB allocator tests...\n\n");
    cprintf("The slab struct size is %d bytes\n", sizeof(slab_t));
    cprintf("----------------------START-------------------------\n");

    // 验证初始化后的对象数量是否正确
    size_t nums[3] = {126, 63, 31};  // 32B, 64B, 128B对应的每个slab对象数
    for (int i = 0; i < cache_n; i++) {
        assert(caches[i].objs_num == nums[i]);
    }

    size_t nr_1 = nr_free;  // 记录初始空闲页数量

    // 1. 边界检查
```

```
    {
        void *obj = slub_alloc_obj(0);
        assert(obj == NULL);

        obj = slub_alloc_obj(256);  // 超过最大缓存大小
        assert(obj == NULL);

        cprintf("Boundary check passed.\n");
    }

    // 2. 基本分配/释放功能检查
    {
        void *obj1 = slub_alloc_obj(32);
        assert(obj1 != NULL);
        cprintf("Allocated 32-byte object at %p\n", obj1);

        // 验证内存写入
        memset(obj1, 0x66, 32);
        for (int i = 0; i < 32; i++) {
            assert(((unsigned char *)obj1)[i] == 0x66);
        }
        cprintf("Memory alloc verification passed.\n");

        // 释放对象
        slub_free_obj(obj1);

        // 验证释放后重新分配的内存是否清零
        void *obj2 = slub_alloc_obj(32);
        cprintf("Allocated 32-byte object at %p\n", obj2);
        for (int i = 0; i < 32; i++) {
            assert(((unsigned char *)obj2)[i] == 0x00);
        }
        slub_free_obj(obj2);

        cprintf("Memory free verification passed.\n");
    }

    // 3. 多个对象分配/释放检查
    {
        const int NUM_TEST_OBJS = 10;
        void *test_objs[NUM_TEST_OBJS];

        cprintf("Allocating %d objects of size 64 bytes.\n", NUM_TEST_OBJS);
        for (int i = 0; i < NUM_TEST_OBJS; i++) {
            test_objs[i] = slub_alloc_obj(64);
            assert(test_objs[i] != NULL);
            memset(test_objs[i], i, 64);  // 每个对象写入不同的值
        }

        // 验证内存内容
        for (int i = 0; i < NUM_TEST_OBJS; i++) {
            for (int j = 0; j < 64; j++) {
                assert(((unsigned char *)test_objs[i])[j] == (unsigned char)i);
            }
        }
        cprintf("Memory verification for 64-byte objects passed.\n");
```

```
    // 释放并验证
    for (int i = 0; i < NUM_TEST_OBJS; i++) {
        slub_free_obj(test_objs[i]);
        cprintf("Freed 64-byte object at %p\n", test_objs[i]);
        // 验证内存已清零
        for (int j = 0; j < 64; j++) {
            assert(((unsigned char *)test_objs[i])[j] == 0x00);
        }
    }
    cprintf("Memory free verification for 64-byte objects passed.\n");
}


// 4. 混合分配释放流程检查
{
    cprintf("Mixed allocation/free check start.\n");

    // 分配不同大小的对象
    void *obj1 = slub_alloc_obj(32);
    assert(obj1 != NULL);
    cprintf("Allocated 32-byte object at %p\n", obj1);
    assert(nr_free == nr_1 - 1);

    void *obj2 = slub_alloc_obj(64);
    assert(obj2 != NULL);
    cprintf("Allocated 64-byte object at %p\n", obj2);
    assert(nr_free == nr_1 - 2);

    void *obj3 = slub_alloc_obj(128);
    assert(obj3 != NULL);
    cprintf("Allocated 128-byte object at %p\n", obj3);
    assert(nr_free == nr_1 - 3);

    // 再分配一个32字节对象，应该使用同一个Slab
    void *obj4 = slub_alloc_obj(32);
    assert(obj4 != NULL);
    cprintf("Allocated second 32-byte object at %p\n", obj4);
    assert(nr_free == nr_1 - 3);

    // 分配29个128字节对象(当前Slab共30个)
    void *objs[30];
    for (int i = 0; i < 29; i++) {
        objs[i] = slub_alloc_obj(128);
        assert(objs[i] != NULL);
    }

    // 第31个128字节对象，应该还在同一个Slab
    void *obj5 = slub_alloc_obj(128);
    assert(obj5 != NULL);
    cprintf("Allocated 31th 128-byte object at %p\n", obj5);
    assert(nr_free == nr_1 - 3);

    // 第32个128字节对象，需要新的Slab
    void *obj6 = slub_alloc_obj(128);
    assert(obj6 != NULL);
```

```
        cprintf("Allocated 32th(new slab) 128-byte object at %p\n", obj6);
        assert(nr_free == nr_1 - 4);

        // 释放29个128字节对象
        for (int i = 0; i < 29; i++) {
            slub_free_obj(objs[i]);
        }
        assert(nr_free == nr_1 - 4);

        // 逐步释放对象，验证内存回收
        slub_free_obj(obj1);
        assert(nr_free == nr_1 - 4);

        slub_free_obj(obj2);
        assert(nr_free == nr_1 - 3);

        slub_free_obj(obj3);
        assert(nr_free == nr_1 - 3);

        slub_free_obj(obj4);
        assert(nr_free == nr_1 - 2);

        slub_free_obj(obj5);
        assert(nr_free == nr_1 - 1);

        slub_free_obj(obj6);
        assert(nr_free == nr_1);  // 回到初始状态

        cprintf("Mixed allocation/free check passed.\n");
    }

    cprintf("----------------------END-------------------------\n");
    cprintf("All SLUB allocator tests passed successfully!\n");
}
```

与ucore集成

同buddy_system逻辑类似，不做赘述。

```
#ifndef __KERN_MM_SLUB_PMM_H__
#define __KERN_MM_SLUB_PMM_H__

#include <pmm.h>

extern const struct pmm_manager slub_pmm_manager;

#endif
```

测试结果

同样输入make clean && make -j:

```
ubuntu@Ubuntu-lj:~/Desktop/lab2/lab2$ make clean && make -j
rm -f -r obj bin
+ 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/buddy_pmm.c
+ cc kern/mm/best_fit_pmm.c
+ cc kern/mm/default_pmm.c
+ cc kern/mm/pmm.c
+ cc kern/mm/slub_pmm.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 bin/ucore.img
```

输入make qemu:

```
----------------------START-------------------------
Boundary check passed.
Allocated 32-byte object at 0xffffffffc0347028
Memory alloc verification passed.
Allocated 32-byte object at 0xffffffffc0347028
Memory free verification passed.
Allocating 10 objects of size 64 bytes.
Memory verification for 64-byte objects passed.
Freed 64-byte object at 0xffffffffc0347028
Freed 64-byte object at 0xffffffffc0347068
Freed 64-byte object at 0xffffffffc03470a8
Freed 64-byte object at 0xffffffffc03470e8
Freed 64-byte object at 0xffffffffc0347128
Freed 64-byte object at 0xffffffffc0347168
Freed 64-byte object at 0xffffffffc03471a8
Freed 64-byte object at 0xffffffffc03471e8
Freed 64-byte object at 0xffffffffc0347228
Freed 64-byte object at 0xffffffffc0347268
Memory free verification for 64-byte objects passed.
Mixed allocation/free check start.
Allocated 32-byte object at 0xffffffffc0347028
Allocated 64-byte object at 0xffffffffc0348028
Allocated 128-byte object at 0xffffffffc0349028
Allocated second 32-byte object at 0xffffffffc0347048
Allocated 31th 128-byte object at 0xffffffffc0349f28
Allocated 32th(new slab) 128-byte object at 0xffffffffc034a028
Mixed allocation/free check passed.
----------------------END---------------------------
```

根据输出可知我们的测试顺利通过。

扩展练习三 硬件的可用物理内存范围的获取方法

核心思想

如果 OS 无法提前知道可用的物理内存范围，它必须依赖以下核心思想：

依赖底层固件和引导加载程序提供的标准接口。

核心流程为：

1. **硬件固件**进行底层探测，获取内存的物理布局。

2. **引导加载程序**作为中介，获取这个布局信息。
3. **引导加载程序**以一种标准化的方式（如寄存器传参、传递结构体地址）将内存映射表交给 **OS 内核。**
4. **OS 内核**解析该表，标记出所有可用的 RAM 区域，并将其交给物理内存管理器进行分配。

具体方案

方案一：基于设备树（DTB）的机制（RISC-V/ARM 架构）

在以 **RISC-V** 和 **ARM** 为代表的嵌入式和现代服务器架构中，硬件配置是高度抽象化的，其内存映射信息通过 DTB 传递。

步骤	机制	描述
固件加载	**DTB (Device Tree Blob)**	硬件制造商提供一个设备树文件，其中包含所有硬件资源的描述，特别是 `/memory` 节点，精确定义了所有内存块的起始地址和大小。
信息传递	**Bootloader 传递**	**Bootloader**（如 OpenSBI）加载 DTB，并在跳转到内核入口点之前，通过预定义的寄存器（例如 RISC-V 上的 `a1` 寄存器）将 DTB 结构的**物理地址**传递给内核。
内核解析	**内存映射初始化**	OS 内核启动后，读取寄存器获取 DTB 地址，解析 `/memory` 节点。内核根据解析出的信息来初始化 `struct Page` 数组和物理内存管理器。

通俗地说，硬件厂商已经事先把内存配置信息写死在一个 清单（DTB） 里，**引导程序**把这个清单交给 **OS**，OS 照着清单来管理内存

方案二：基于固件服务的机制（传统 x86/x64 架构）

在 x86 平台上，获取内存映射信息依赖于传统的 **BIOS 中断**或现代的 **UEFI 运行时服务。**

机制	类型	描述
传统 BIOS 中断	`INT 15h, AX=E820h`	在实模式下，OS 调用 **BIOS 中断** 请求内存映射。BIOS 返回一个 **E820 内存映射表**，其中列出了所有物理内存区域的起始地址、大小和类型。
现代 UEFI 服务	`GetMemoryMap`	在 UEFI 固件环境中，**Bootloader** 调用 UEFI 提供的 `GetMemoryMap` 服务。固件返回一个详细的内存描述符列表，其中包含所有内存块（包括可用内存、固件保留内存等）的精确信息。

通俗地说，在传统 x86 架构里，OS 不能直接知道内存，它必须依赖 **Bootloader** 去呼叫底层的 **BIOS/UEFI 固件**，让固件在运行时动态生成并提供一个**“实时内存状态表”**，OS 拿到这个表后才能开始工作。

知识点

实验重要知识点与对应os知识点

1.连续内存分配算法

实验中实现了两种连续内存分配策略：

- First-Fit：从空闲链表头部开始遍历，找到第一个大小满足需求的空闲块即分配，拆分剩余部分并更新链表。
- Best-Fit：遍历整个空闲链表，选择大小满足需求的最小空闲块分配，同样涉及块拆分与链表维护。两种算法均通过双向循环链表按地址升序管理空闲块，释放时需按地址插入并合并相邻块以减少碎片。

对应os原理知识点：

- 连续内存分配策略（首次适配、最佳适配等）。原理中重点介绍不同策略的核心思想：如何从空闲分区中选择合适的块分配给进程，以及策略对分配效率和碎片产生的影响。

2. 内存碎片与合并

- **实验知识点**：释放内存时，通过 “前向合并”和 “后向合并”消除外部碎片。例如，`default_free_pages` 和 `best_fit_free_pages` 中，通过地址连续性判断（如 `p + p->property == base`）合并相邻块，更新块大小并删除冗余链表节点。
- **对应 OS 原理知识点**：外部碎片的概念（已分配块之间的小空闲块，无法满足新的分配请求）及碎片整理方法（合并相邻空闲块、内存紧凑等）。

3. 伙伴系统（Buddy System）

- **实验知识点**：基于 “2 的幂次块” 管理内存：
  - 分配时将请求大小向上取整为最近的 2^n 页，若无对应阶数空闲块，则拆分高阶块直至匹配。
  - 释放时自动检测相邻 “伙伴块”（地址连续、大小相同），合并为更高阶块，直至无法合并。用 `free_list[MAX_ORDER]` 数组（按阶数划分）管理空闲块，通过位运算快速定位伙伴块。
- **对应 OS 原理知识点**：伙伴系统算法。原理中说明其核心是通过 “块的拆分与合并” 高效管理内存，适合频繁分配 和释放不同大小块的场景，平衡了分配效率与碎片问题。

4. 物理页框状态管理

- **实验知识点**：通过 `struct Page` 结构体管理物理页状态：
  - `flags`：标记页是否为 “空闲块起始页”（`PG_property`）、是否为保留页（`PG_reserved`）等。
  - `property`：记录空闲块大小（连续分配算法）或阶数（伙伴系统）。
  - 引用计数（`ref`）：记录页被进程引用的次数，确保释放时的安全性。
- **对应 OS 原理知识点**：物理内存的页框管理。原理中说明操作系统需跟踪每个页框的状态（是否空闲、被哪个进程使用、是否被修改等），以实现内存分配与回收。

实验中未涉及知识点

1. **页面置换算法**

   原理中当物理内存不足时，需通过 LRU（最近最少使用）、FIFO 等算法将不常用页面换出到磁盘，以腾出空间。实验中内存分配仅判断 “空闲页是否足够”，无内存不足时的置换逻辑。

2. **内存保护机制**

   原理中通过页表项的权限位（读 / 写 / 执行）实现内存保护，防止进程越权访问（如用户进程修改内核内存）。实验未涉及权限管理，所有页的访问权限未做区分。

3. **分段与段页式管理**

   原理中分段管理按程序逻辑划分地址空间（如代码段、数据段），段页式结合分段与分页的优点。实验仅涉及连续物理内存分配，未涉及逻辑地址的分段或段页式映射。