LAB2

练习1：理解first-fit 连续物理内存分配算法

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。

首先，我们需要了解其数据结构：

```
typedef struct {
    list_entry_t free_list;  // 双向链表，存储空闲块的链表节点
    size_t nr_free;          // 当前空闲块的总数
} free_area_t;

free_area_t free_area;

#define free_list (free_area.free_list)    // 链表头
#define nr_free (free_area.nr_free)        // 空闲块数量
```

free_area_t用于管理空闲块链表，struct Page用于定义页结构代表一个物理页，链表中的每个节点对应一个物理页块。

default_init()是初始化函数，用于初始化空闲链表，并将空闲页数量设置为零。default_init_memmap()将一段内存块初始化为空闲块，并插入空闲链表，并初始化每个页的标志位和引用计数，设置块的大小并更新空闲页数量：

```
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;  // 清除标志和属性
        set_page_ref(p, 0);          // 引用计数设置为 0
    }
    base->property = n;              // 第一个页块的属性设置为块大小
    SetPageProperty(base);           // 标记该页块为有效页块
    nr_free += n;                    // 增加空闲页数量

    // 将块插入空闲链表
    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;
            }
        }
    }
}
```

传入的参数为指向这块内存区域的起始页（第一个页面的 struct Page）指针，和该块中包含的页数n（即内存大小）。如果 n 为 0，说明没有任何页面需要初始化，程序会触发 断言（assert）错误，否则进行遍历初始化操作，确保页面当前是已保留状态（即尚未被使用）后将标志位、属性和引用计数均设置为0。接下来检查链表是否为空：如果 `free_list` 链表为空，直接将块插入链表中作为第一个块。否则，遍历链表，找到适当的插入位置：使用 `list_next(le)` 遍历链表的每个节点，`le2page()`是一个宏，用于将链表节点转换为页面结构 struct Page。如果 base 小于当前页面，则在当前节点之前插入新块。这个流程确保了每次插入后的链表**始终按地址升序排列**，方便后续的内存分配和合并操作。

`default_alloc_page` 实现了一个物理内存管理器中的首次适应算法（First-Fit Allocation），用于分配请求的内存页块。它从空闲页链表中找到第一个足够大的块来满足请求，并根据请求的大小进行分割、调整，更新链表和空闲页数量。

```
static struct Page *default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) return NULL;  // 如果没有足够的空闲页，返回 NULL

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

    // 遍历空闲链表，找到第一个满足大小的块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);// 将链表节点转换为页面结构
        if (p->property >= n) {  // 找到合适的块
            page = p;
            break;
        }
    }

    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));  // 从链表中删除已分配的块

        if (page->property > n) {  // 如果块的大小大于请求大小
            struct Page *p = page + n;  // 计算剩余部分的起始页面
            p->property = page->property - n;  // 设置剩余块的大小
            SetPageProperty(p);  // 标记剩余块为有效块
            list_add(prev, &(p->page_link));  // 将剩余块重新插入链表
        }


        nr_free -= n;  // 更新空闲页数量
        ClearPageProperty(page);  // 清除分配块的属性
    }
```

```
    return page;
}
```

`default_free_pages` 实现了内存管理器中的页面释放逻辑，将已使用的页面块重新释放回空闲链表，并尝试合并相邻块，以合并前一个块为例，代码如下：

```
list_entry_t* le = list_prev(&(base->page_link));
if (le != &free_list) {
    p = le2page(le, page_link);
    if (p + p->property == base) {  // 判断是否与前一个块相邻
        p->property += base->property;  // 合并块的大小
        ClearPageProperty(base);  // 清除被合并块的属性
        list_del(&(base->page_link));  // 从链表中删除被合并的块
        base = p;  // 更新 base 指向新的合并后的块
    }
}
```

在了解完代码的基本含义后，我们需要讨论：

- first fit算法是否有进一步的改进空间？ first fit算法的优点是实现简单，缺点是会产生很多小的空闲块，导致内存碎片化严重。可以考虑加入碎片化的内存整理算法，将多个小的空闲块合并成一个大的空闲块，从而减少内存碎片化。

练习2：实现 Best-Fit 连续物理内存分配算法

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?

在该实验中，我们基于 `First-Fit` 的页面分配算法，修改并实现了 `Best-Fit` 分配算法。Best-Fit 算法的核心思想是：在空闲内存块中寻找**最小的满足请求的块**，从而减少内存的浪费和碎片。以下是代码实现的主要步骤与思路。

核心代码实现

1. **初始化 (`best_fit_init`)**
    - 初始化空闲链表 `free_list`，并将空闲块计数 `nr_free` 置为 0。
2. **初始化内存映射 (`best_fit_init_memmap`)**
    - 遍历物理页框，清除每页的标志和引用计数。
    - 将新分配的空闲页块插入到空闲链表中，按地址顺序排列。
3. **分配页面 (`best_fit_alloc_pages`)**
    - 遍历 `free_list`，找到**大小刚好满足请求的最小块**。

- 如果找到适配的块，将其从空闲链表中移除，并更新剩余块的大小。
- 若无满足请求的块，则返回 `NULL`。

4. **释放页面 (`best_fit_free_pages`)**
   - 将释放的块重新插入到 `free_list`，并按地址顺序排列。
   - 如果相邻块可以合并，则执行合并操作，并更新块大小。

5. **测试函数 (`best_fit_check`)**
   - 包含一系列测试用例，检查分配与释放的正确性，以及是否实现了 Best-Fit 的特性。

实现过程中的关键逻辑

- **空闲链表管理：**
  - 使用双向链表 (`list`) 存储空闲页面块。
  - 插入新块时，按地址顺序插入，方便后续的合并操作。
- **最小块查找：**
  - 在分配页面时，遍历 `free_list`，找到**大小刚好满足需求的最小块**。
  - 避免不必要的内存浪费。
- **合并相邻块：**
  - 在释放页面时，检查前后相邻的块是否可以合并。
  - 如果相邻块可以合并，则更新其大小，并将其标记为单个连续块。

代码是如何进行分配和释放的

1. **分配流程**：
   - 遍历空闲链表，找到满足需求的最小块。
   - 若找到的块比需求大，将剩余部分分为新块，并重新加入链表。
   - 将分配的页面块从链表中移除，并标记为已分配状态。
2. **释放流程**：
   - 将释放的块重新插入空闲链表，并按地址排序。
   - 如果前后块与当前块相邻，则进行合并，并更新合并后的块大小。

代码的改进空间

1. **时间复杂度优化**：
   - 当前实现采用线性遍历寻找最小块，复杂度为 `O(n)`。可以考虑使用 **平衡二叉树** 或 **最小堆** 来管理空闲块，从而将查找操作优化为 `O(log n)`。
2. **内存碎片问题**：

- 如果页面释放和分配频繁发生，可能会出现较多的内存碎片。可以进一步优化合并策略，例如在空闲块数量较多时触发碎片整理。

3. **多线程环境支持**：
   - 在多线程环境中，链表操作可能引发竞态条件。可以考虑引入锁机制或采用**无锁数据结构**以支持多线程安全。

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。

```
struct buddy2 {
    int size;                // 内存总大小
    int *longest;            // 节点数组，存储可分配的最大块大小
};
```

- `size` 表示整个分配区域的大小。
- `longest` 是一个数组，`longest[i]` 存储第 i 个节点可分配的最大内存块。

```
void buddy_init(struct buddy2 *self, int size) {
    self->size = size;
    self->longest = malloc(2 * size * sizeof(int));
    for (int i = 0; i < 2 * size - 1; ++i) {
        self->longest[i] = pow(2, (int)(log2(size)) - (int)(log2(i + 1)));
    }
}
```

1. 初始化完全二叉树的节点信息。
2. 为每个节点分配初始最大内存大小。

分配内存

```
int buddy_alloc(struct buddy2 *self, int size) {
    size = pow(2, ceil(log2(size)));  // 将大小调整为 2 的幂
    int index = 0;

    while (index < self->size - 1) {
        if (self->longest[index * 2 + 1] >= size) {
            index = index * 2 + 1;  // 选择左子树
        } else if (self->longest[index * 2 + 2] >= size) {
            index = index * 2 + 2;  // 选择右子树
        } else {
```

```
                return -1;  // 没有可用块
            }
        }

        self->longest[index] = 0;  // 将该块标记为已分配
        int offset = (index + 1) * size - self->size;
        return offset;
    }
```

- 搜索适配的内存块并分配给用户。
- 将分配的块大小标记为 0，表示已使用。

释放内存

```
void buddy_free(struct buddy2 *self, int offset, int size) {
    int index = offset + self->size - 1;

    self->longest[index] = size;  // 恢复块大小
    while (index > 0) {
        index = (index - 1) / 2;
        self->longest[index] = fmax(
            self->longest[index * 2 + 1],
            self->longest[index * 2 + 2]
        );
    }
}
```

- 找到释放块的节点并恢复其状态。
- 检查左右子节点是否可以合并，若可以则更新父节点状态。

验证 Buddy System

```
void alloc_check(struct buddy2 *self) {
    // 自定义测试用例，检查分配和释放的正确性
}
```

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。

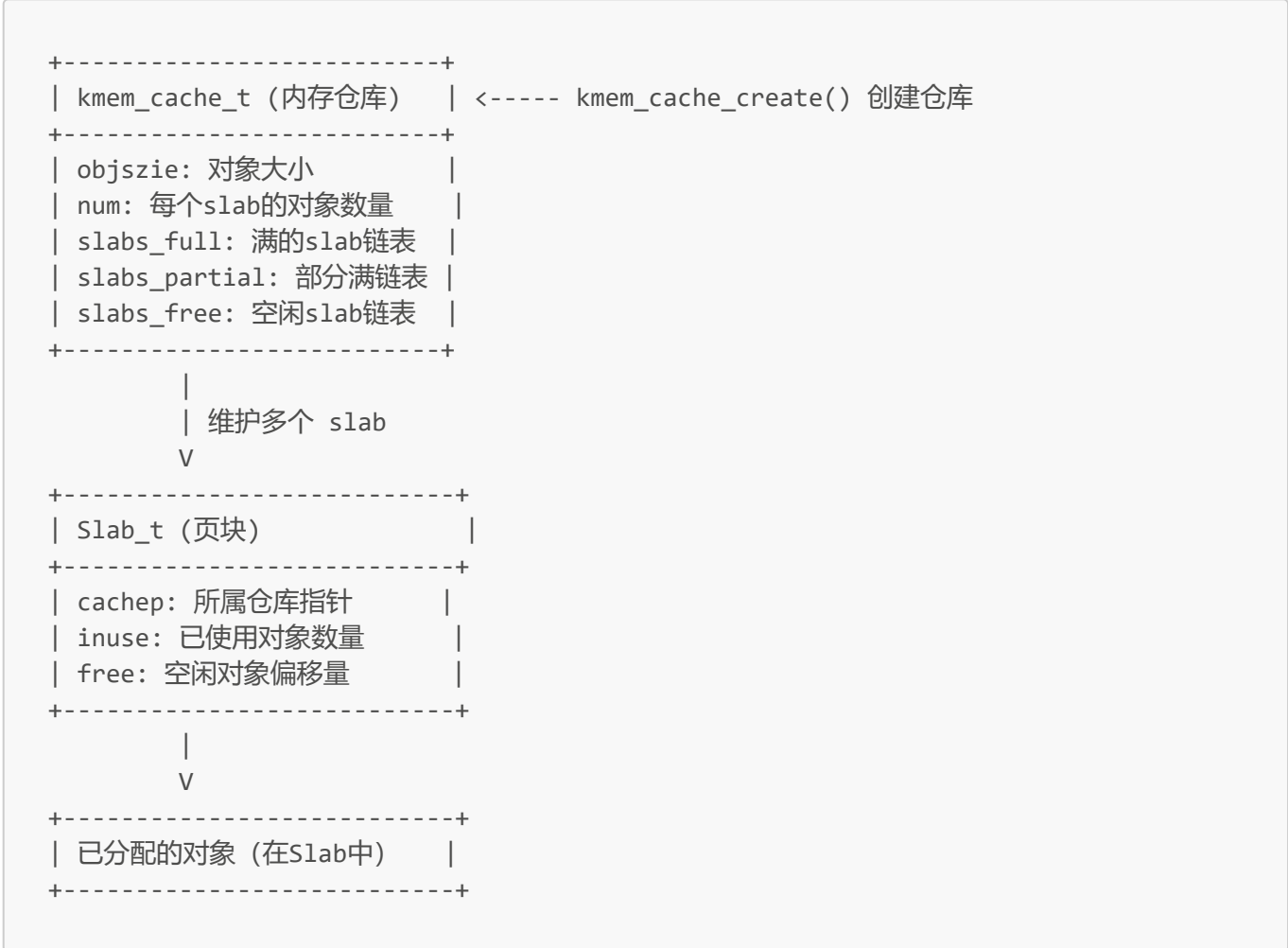
在操作系统中，频繁的小块内存分配和释放会导致 **内存碎片**，降低系统性能。为了解决这个问题，SLUB 分配器对内存进行 **分类和缓存**，以便快速分配小块内存，提高性能并减少碎片。

- **SLUB** 是一种 **Slab 分配器**的实现，广泛用于 **Linux 内核**和嵌入式操作系统中。
- **Slab 分配器**的目标是高效地管理 **同类型的小块内存**，将频繁使用的对象存储在 **内存仓库（kmem_cache）** 中进行缓存，减少分配和释放的开销。

为实现slub算法，我们引入了如下概念：

1. **Slab（页块）：**
    - **Slab** 是一块 **物理页面（通常是4KB大小）**，每个 slab 存储多个相同类型的对象。
    - 每个对象的大小是固定的，比如 64 字节、128 字节等。
    - 一个 slab 包含的对象数量 = 页大小 / 对象大小。
2. **Kmem Cache（内存仓库）：**
    - **内存仓库** 是用于存储特定类型对象的集合，每个仓库由多个 **slab** 组成。
    - 每种对象类型（例如 `struct inode` 或 `struct dentry`）有自己专属的 **kmem_cache**。
    - **仓库** 通过缓存已经分配过的对象，加快后续的分配速度。

基于上述内容，我们设计出了一个简单的slub分配算法的架构：



代码实现

这段代码实现了 **SLUB（Simple and Fast Slab Allocator）内存分配器**。SLUB 是一种高效的内存管理算法，用于动态分配和管理内存块。它通过将内存分为 **slab**（页块）进行管理，每个 slab 中存储一组大小一致的对象。SLUB 适用于频繁的小块内存分配和释放，并通过伙伴系统（buddy system）从物理内存中申请页。

1. `slab_t`：描述单个 Slab 的结构，每个 slab 是一块页面，并包含多个相同大小的对象。

```
struct slab_t {
    int ref;                        // 页的引用次数
    struct kmem_cache_t *cachep; // 关联的内存仓库
    uint16_t inuse;                 // 已分配的对象数量
    int16_t free;                   // 下一个空闲对象的偏移
    list_entry_t slab_link;         // 链接到对应的slab链表
};
```

2. `kmem_cache_t`：内存仓库的结构，用于管理多个 slab。

```
struct kmem_cache_t {
    size_t objsize;                 // 对象大小
    int num;                        // 每个slab中的对象数量
    void (*ctor)(void *, struct kmem_cache_t *, size_t);  // 构造函数
    void (*dtor)(void *, struct kmem_cache_t *, size_t);  // 析构函数
    char name[CACHE_NAMELEN];       // 仓库名称
    list_entry_t slabs_full;        // 满的slab链表
    list_entry_t slabs_partial;  // 部分满的slab链表
    list_entry_t slabs_free;        // 空闲的slab链表
    list_entry_t cache_link;        // 链接到 cache_chain 链表
};
```

3. 从伙伴系统申请页面并初始化 Slab（`kmem_cache_grow`）

- 申请一个页，并将其初始化为一个新的 slab。每个对象按顺序存放在该页中，构建一个静态链表记录空闲对象的偏移。
- 初始化完毕后，将 slab 加入到空闲 slab 链表中。

```
static void *kmem_cache_grow(struct kmem_cache_t *cachep) {
    struct Page *page = alloc_page();    // 从伙伴系统中申请一个页
    void *kva = page2kva(page);          // 将页转为虚拟地址
    struct slab_t *slab = (struct slab_t *)page;
    slab->cachep = cachep;
    slab->inuse = slab->free = 0;

    int16_t *bufctl = kva;
    for (int i = 1; i < cachep->num; i++)
        bufctl[i - 1] = i;
    bufctl[cachep->num - 1] = -1;

    void *buf = bufctl + cachep->num;
    if (cachep->ctor)
        for (void *p = buf; p < buf + cachep->objsize * cachep->num; p += cachep-
>objsize)
            cachep->ctor(p, cachep, cachep->objsize);
```

```
    list_add(&(cachep->slabs_free), &(slab->slab_link));
    return slab;
}
```

4. 释放 Slab 并回收页面（kmem_slab_destroy）：释放一个 slab，调用析构函数处理对象，并将页面归还给伙伴系统。

```
static void kmem_slab_destroy(struct kmem_cache_t *cachep, struct slab_t *slab) {
    struct Page *page = (struct Page *)slab;
    int16_t *bufctl = page2kva(page);
    void *buf = bufctl + cachep->num;
    if (cachep->dtor)
        for (void *p = buf; p < buf + cachep->objsize * cachep->num; p += cachep-
>objsize)
            cachep->dtor(p, cachep, cachep->objsize);
    page->property = page->flags = 0;
    list_del(&(page->page_link));
    free_page(page);  // 归还给伙伴系统
}
```

5. 分配对象（kmem_cache_alloc）:在部分满的 slab 中查找空闲位置，若没有则从空闲的 slab 中分配。若所有 slab 已满，则从伙伴系统中申请新的 slab。最后，更新 slab 的元信息。

```
void *kmem_cache_alloc(struct kmem_cache_t *cachep) {
    list_entry_t *le = list_empty(&(cachep->slabs_partial)) ?
                       list_next(&(cachep->slabs_free)) : list_next(&(cachep-
>slabs_partial));

    struct slab_t *slab = le2slab(le, page_link);
    void *kva = slab2kva(slab);
    int16_t *bufctl = kva;
    void *buf = bufctl + cachep->num;
    void *objp = buf + slab->free * cachep->objsize;

    slab->inuse++;
    slab->free = bufctl[slab->free];

    if (slab->inuse == cachep->num)
        list_add(&(cachep->slabs_full), le);
    else
        list_add(&(cachep->slabs_partial), le);

    return objp;
}
```

6. 释放对象：找到对象所在的 slab，并将其标记为未使用。如果 slab 变空，则将其移至空闲 slab 链表中。

```
void kmem_cache_free(struct kmem_cache_t *cachep, void *objp) {
    void *kva = ROUNDDOWN(objp, PGSIZE);
    struct slab_t *slab = (struct slab_t *)&pages[(kva - slab2kva(pages)) /
PGSIZE];

    int16_t *bufctl = kva;
    void *buf = bufctl + cachep->num;
    int offset = (objp - buf) / cachep->objsize;

    list_del(&(slab->slab_link));
    bufctl[offset] = slab->free;
    slab->inuse--;
    slab->free = offset;

    if (slab->inuse == 0)
        list_add(&(cachep->slabs_free), &(slab->slab_link));
    else
        list_add(&(cachep->slabs_partial), &(slab->slab_link));
}
```

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?

1. **获取设备树地址**：在内核启动时，可以通过 RISC-V 的约定，从 a1 寄存器中获取设备树的物理地址。在汇编启动代码中获取这个地址并传递给主内核。
2. **解析设备树格式**： 设备树数据是一个树形结构，描述了硬件的各种节点。这个数据以二进制格式（DTB, Device Tree Blob）存储，通常包括节点的以下信息：

- 节点名称（例如 /memory 表示物理内存）
- 地址范围属性（reg），通常包含起始地址和内存大小
- 其他硬件属性（如 CPU、外设等）
- 可以使用已有的设备树解析库，如 Linux 中的 libfdt（Flattened Device Tree），逐层遍历设备树节点，找到 memory 节点的 reg 属性值，获得内存地址和大小。

3. **解析 /memory 节点**： 通过设备树根节点开始遍历，查找名为 /memory 的节点。这一节点通常包含 reg 属性，用来定义内存的起始地址和大小。
4. **使用解析出的地址信息**： 通过解析获得的内存地址和大小信息，操作系统可以初始化内存管理单元，设置页面表等。