OSLab2操作系统的物理内存管理

1. 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。 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?

1.1. 算法理解与实现过程描述

First-fit的基本思想

> **从头开始遍历空闲块链表，找到第一个大小足以满足请求的空闲块，并进行分配。**

实现过程描述

1. `default_init()`

- 功能：初始化默认的物理内存管理器。
- 通常在 `init_pmm_manager()` 被调用时执行。
- 操作：
  - 初始化空闲块链表 `free_list`。
  - 将空闲页数量 `nr_free` 设为 0。

2. `default_init_memmap(struct Page *base, size_t n)`

- 功能：将一段连续的内存页初始化为一个空闲块，并插入到空闲链表free_list中。
- 实现逻辑：
  - 在确认传入的n大于0后，函数遍历从base开始的n个页，确保每一页最初处于"保留"状态，同时清除当前页的flags和property字，并将每个页的引用计数设为 0，完成初始化。
  - 对第一个页进行修改，设置其PG_property标志位，以表示其是头部页，同时设置第一个页的property字段，以表示该空闲块大小为n。

- 最后便是依据空闲块链表是否为空，进行新空闲块的插入，如果链表非空，通过函数list_next(le)遍历空闲链表，寻找合适的位置插入新的空闲块。然后if (base < page)判断 base 是否位于当前链表节点之前。如果遍历到链表的末尾都没有找到合适的位置（即base的地址大于当前链表中的所有块），则将 base 插入到链表的末尾。

3. `default_alloc_pages(size_t n)`

- 功能：根据first-fit算法找到合适的空闲块并进行分配。
- 实现过程：
  - 遍历 `free_list`。
  - 找到第一个 `p->property >= n` 的块。
  - 若该块刚好相等，直接移除该节点。
  - 若该块更大，则计算剩余部分的起始地址 p，并在更新剩余部分的property字段后，将剩余部分p插入到链表原位置中。
  - 更新 `nr_free`，并返回最后返回找到并分配的内存块的指针 page。
- 算法复杂度：O(m)，m 为空闲块数量。

4. `default_free_pages(struct Page *base, size_t n)`

- 功能：释放分配的内存，并尝试合并相邻的空闲块，以减少内存碎片。
- 实现过程：
  - 遍历空闲链表，找到插入位置（保持按地址递增）。
  - 检查是否可以与前后块合并（相邻则合并）。
  - 更新 `nr_free`。

1.2. 思考题回答

First-Fit的主要问题：

- **外部碎片问题**：随着分配和释放的进行，前端小块空闲内存累积，导致大块内存难以分配。
- **分配速度下降**：随着free_list增长，查找第一个合适块可能遍历整个列表，效率下降。
- **分配不均衡**：总是从列表前端开始分配，会让前端的内存频繁分配和释放，后端的空闲内存长时间闲置。

---

2. 改进方法

（1）Best-Fit或Worst-Fit算法

- **Best-Fit**：选择最小但足够大的空闲块分配。

- **优点**：减小浪费，减少外部碎片。
- **缺点**：可能产生大量很小的碎片。

（2）空闲块索引结构优化

- **链表改进**：
  - 按大小排序，分配更快（如按大小升序）。
  - 使用多个链表管理不同大小的空闲块（分级空闲链表）。
- **树形结构**：
  - 使用平衡树（如红黑树）按大小管理空闲块，查找合适块速度更快（O(log n)）。

（3）缓存/延迟分配

- **延迟释放**：对于频繁申请释放的小块，先缓存再统一处理，减少链表操作开销。
- **缓存常用大小块**：如SLAB分配器思想，提升分配效率。

2. 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?

2.1. 算法设计与实现过程

Best-fit算法思想及原理

> best-fit的思想是遍历整个空闲链表，选择**最小但仍能满足需求的空闲块**进行分配。即在first fit算法的基础上，于空闲块链表中寻找满足所需内存大小的最小内存块，其余步骤与first fit算法基本相同。

设计实现

1、best-fit与first-fit的内存页初始化是一致，都是将一段连续的内存页初始化为一个空闲块，并插入到空闲链表free_list中，故best_fit_init_memmap代码与default_init_memmap应当一致。

代码块

```
1  static void
2  best_fit_init_memmap(struct Page *base, size_t n) {
3      assert(n > 0);
4      struct Page *p = base;
5      for (; p != base + n; p ++) {
```

```
            assert(PageReserved(p));
            /*LAB2 EXERCISE 2: YOUR CODE*/
            // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
            //初始化每个页
            p->flags = p->property = 0;
            set_page_ref(p, 0);
        }
        base->property = n;
        SetPageProperty(base);
        nr_free += n;
        if (list_empty(&free_list)) {
            list_add(&free_list, &(base->page_link));
        } else {
            list_entry_t* le = &free_list;
            while ((le = list_next(le)) != &free_list) {
                struct Page* page = le2page(le, page_link);
                /*LAB2 EXERCISE 2: YOUR CODE*/
                // 编写代码
                // 1、当base < page时，找到第一个大于base的页，将base插入到它前面，并退出循环
                // 2、当list_next(le) == &free_list时，若已经到达链表结尾，将base插入到链表尾部
                if (base < page) {
                    list_add_before(le, &(base->page_link));
                    break;
                }
                //如果全部块地址都比base小,则放在最后
                else if (list_next(le) == &free_list) {
                    list_add(le, &(base->page_link));
                }
            }
        }
    }
```

2、first-fit与best-fit最大的不同体现在分配算法上。代码部分相对first-fit最关键的在于需要利用一个page和一个minsize变量，在遍历空闲链表时始终记录当前满足分配条件的最小块的地址和大小（原则：最小但够大）。在遍历结束后，使用这个合适的最小块进行分配。

代码块

```
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
```

```
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
    /*LAB2 EXERCISE 2: YOUR CODE*/
    // 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);
        // 这个块够大才考虑
        if (p->property >= n) {
            // 若这是第一个合适的块，或比当前最小块还小
            if (p->property < min_size) {
                min_size = p->property;
                page = p;
            }
        }
    }

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

3、best-fit与first-fit的内存释放是一致，都是释放分配的内存，并尝试合并相邻的空闲块，以减少内存碎片。故代码一致。

代码块

```
static void
best_fit_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
```

```
7           p->flags = 0;
8           set_page_ref(p, 0);
9       }
10      /*LAB2 EXERCISE 2: YOUR CODE*/
11      // 编写代码
12      // 具体来说就是设置当前页块的属性为释放的页块数、并将当前页块标记为已分配状态、最后增加nr_free的值
13      base->property = n;
14      SetPageProperty(base);
15      nr_free += n;
16
17      if (list_empty(&free_list)) {
18          list_add(&free_list, &(base->page_link));
19      } else {
20          list_entry_t* le = &free_list;
21          while ((le = list_next(le)) != &free_list) {
22              struct Page* page = le2page(le, page_link);
23              if (base < page) {
24                  list_add_before(le, &(base->page_link));
25                  break;
26              } else if (list_next(le) == &free_list) {
27                  list_add(le, &(base->page_link));
28              }
29          }
30      }
31
32      list_entry_t* le = list_prev(&(base->page_link));
33      if (le != &free_list) {
34          p = le2page(le, page_link);
35          /*LAB2 EXERCISE 2: YOUR CODE*/
36          // 编写代码
37          // 1、判断前面的空闲页块是否与当前页块是连续的，如果是连续的，则将当前页块合并到前面的空闲页块中
38          // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
39          // 3、清除当前页块的属性标记，表示不再是空闲页块
40          // 4、从链表中删除当前页块
41          // 5、将指针指向前一个空闲页块，以便继续检查合并后的连续空闲页块
42          if (p + p->property == base) {
43              p->property += base->property;
44              ClearPageProperty(base);
45              list_del(&(base->page_link));
46              base = p;
47          }
48      }
49
50      le = list_next(&(base->page_link));
51      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));
            }
        }
    }
```

4、测试结果

2.2. 思考题回答

你的Best-Fit算法是否有进一步的改进空间?

问题:

- **产生大量小碎片:**
  - 分配后剩余空间可能太小，无法再分配给其他请求。
  - 长时间运行后，小碎片累积影响内存利用。
- **分配速度慢:**
  - 每次分配都需要遍历整个空闲块链表，查找最小合适块。
  - 空闲块数量大时，开销明显。
- **链表管理复杂:**
  - 频繁拆分小块，链表碎片化严重。

---

2. 改进方法

（1）优化查找速度

- **使用索引结构:**
  - 按空闲块大小建立平衡树（红黑树）或多级链表（分级空闲链表）。
  - 查找最小足够块时间复杂度可降到 O(log n) 或 O(1)（多级链表）。

- **分级链表**：
  - 空闲块按大小分段，每段用链表管理，减少遍历量。
  - 常用于现代内核和内存分配器（如 SLAB、Buddy 系统）。

（2）减少碎片

- **延迟分配或缓存小块**：
  - 对频繁申请释放的小块，先缓存再统一处理，减少碎片。

（3）混合/动态策略

- **Best-Fit + Buddy System**：
  - 空闲块按 2 的幂次划分，分配近似最优块。
- **智能选择策略**：
  - 根据系统负载动态选择 First-Fit / Best-Fit / Next-Fit。比如对大块用 Best-Fit，对小块用 First-Fit，提高速度和空间利用率。

3.扩展练习Challenge1：buddy_system程序设计报告

1. 引言

本报告旨在详细阐述基于伙伴系统（Buddy System）的物理内存管理模块的设计与实现。伙伴系统是一种高效的内存分配算法，它通过将内存块划分为2的幂次方大小，并利用伙伴块的概念进行快速的分配和回收，有效减少了外部碎片，并简化了内存管理。该设计主要实现物理页面的初始化、分配和释放，以及设计检查函数观察buddy_system的物理内存管理逻辑。

2. 设计理念

本实验完成的buddy_system物理内存管理系统，是基于实验提供的pmm初始化框架和物理页初始化框架开发，使用满二叉树为主要的数据结构对物理内存进行管理。

1. 采用模拟C++面对对象的编程方法，绑定pmm_manager到一本伙伴系统实现实例，并且完成结构体中各个虚拟函数的实现。
2. 采用满二叉树的形式管理物理内存，满二叉树采用地址连续的数组而非链表实现，得益于满二叉树的结构，所有非叶子节点都有 左右两个子节点，所有叶子节点都在同一层，可以使用方法便捷访问子节点、父节点。满二叉树数组存储的是当前节点管理的最大空闲空间。

3. 结合页表机制和伙伴系统，以页为分配最小单位，内核可管理内存以页为单位，划分为2的幂次方大小。比如在此次实验中，内核可管理128MB内存，为2的15次方页，但是opensbi和内核加载会消耗部分内存，还有页表数据结构以及buddy_system数据结构会消耗内核end位置后的部分内存，内核实际操纵内存不足128MB，为了防止内核操作空间溢出，选择向下取2的幂次方，即2的14次方。根节点可以操作2的14幂次方页数目，根节点看作第0级，第1级节点可以操纵2的13幂次方页数目，以此类推，知道第12级可以操纵2页，第13级可以操纵1页，即管理最小单位。可见分配和释放操作都是logN的时间复杂度。
4. 基于满二叉树的buddy_system实现，可以说是对满二叉树的数据结构的相关操作的实现。当分配空间时，是以页为单位请求空间的，需要向上取2的幂次方，然后自顶向下搜索符合条件的节点，将其管理的页全部分配给当前的进程。分配之后需要更新该节点的父节点的管理的最大空闲页数据，直到到达某个父亲节点数据不发生变化。当释放时，根据传入的page的起始索引和数目，可以判断释放区域所属的节点位置，进行释放。更新完当前节点后，检查其伙伴块是否也处于空闲状态。如果两个伙伴块都空闲，它们将合并成一个更大的空闲块，这个过程会递归进行，直到不能再合并为止。
5. 可以发现，由于每个空闲块的大小都是2的幂次方，可以根据page的索引，简化查询搜索过程，比如与first_fit相比，释放时需要便利空闲链表，将释放区域插入到正确的物理地址，保证地址有序。

这种设计有以下优点：

· 减少外部碎片：通过合并空闲伙伴块，系统能够形成更大的连续空闲区域，从而减少外部碎片的产生。

· 快速分配与回收：分配时，系统可以直接找到大小合适的空闲块；回收时，通过简单的位运算即可找到伙伴块，并进行合并。

3. 重要数据结构

3.1 pmm_manager

代码块

```
1   struct pmm_manager {
2       const char *name;  // XXX_pmm_manager's name
3       void (*init)(
4           void);  // initialize internal description&management data structure
5                   // (free block list, number of free block) of XXX_pmm_manager
6       void (*init_memmap)(
7           struct Page *base,
8           size_t n);  // setup description&management data structcure according
    to
9                       // the initial free physical memory space
10      struct Page *(*alloc_pages)(
11          size_t n);  // allocate >=n pages, depend on the allocation algorithm
12      void (*free_pages)(struct Page *base, size_t n);  // free >=n pages with
```

```
13                                                      // "base" addr of Page
14                                                      // descriptor
15                                                      //
    structures(memlayout.h)
16      size_t (*nr_free_pages)(void);  // return the number of free pages
17  };
```

模拟C++的面对对象，采用结构体和函数指针实现类的类似功能，本设计实例化了一个buddy_pmm_manager

代码块

```
1   const struct pmm_manager default_pmm_manager = {
2       .name = "default_pmm_manager",
3       .init = default_init,
4       .init_memmap = default_init_memmap,
5       .alloc_pages = default_alloc_pages,
6       .free_pages = default_free_pages,
7       .nr_free_pages = default_nr_free_pages,
8       .check = default_check,
9   };
```

3.2 struct Page

代码块

```
1   struct Page {
2       int ref;                        // page frame's reference counter
3       uint64_t flags;                 // array of flags that describe the status
    of the page frame
4       unsigned int property;          // the num of free block, used in first
    fit pm manager
5       list_entry_t page_link;         // free list link
6   };
```

沿用项目框架下的Page数据结构，但是不再使用page_link、property成员变量，在buddy_system中由于采用满二叉树的数据结构实现，不再需要使用指针，大大简化了项目开发难度。沿用flag信号中的Reserved信号位，但是其实是没有必要的，因为buddy_system中，通过节点管理的最大空闲页数可以反映该页是否可用。沿用ref变量。

3.3 buddy_system的全局变量

代码块

```
// 记录 buddy 系统全局状态
static unsigned buddy_size;              // 可管理页数（必须是 2 的幂）
static unsigned *buddy_longest = NULL;   // 完全二叉树数组，存放管理的最大空闲页数目
static unsigned buddy_node_count;        // = 2 * buddy_size - 1
static struct Page *buddy_base;          // 起始页
static size_t nr_buddy_free_pages;          // 空闲页计数
```

`buddy_longest` 存放的的当前节点管理的最大空间页的数目，在分配和释放时，都需要在修改当前节点 `buddy_longest[index]` 后，向上回溯更新父节点的 `buddy_longest[parent_index]` 。

4. 关键函数实现

4.1 满二叉树的基本操作工具实现

满二叉树寻找父亲节点和左右孩子节点，相比于链表实现的二叉树，可以很好的利用数组和结构特点。节点从索引0开始，当前节点的左孩子为index×2+1，当前节点的右孩子为index×2+2，当前节点的父节点为(index + 1) / 2 - 1。

同时设计父节点更新函数，访问左右孩子的值，选择更大值更新父节点。

代码块

```
#define LEFT_LEAF(index)  ((index) * 2 + 1)//左子节点
#define RIGHT_LEAF(index) ((index) * 2 + 2)//右子节点
#define PARENT(index)     (((index) + 1) / 2 - 1)//父节点
#define MAX(a, b)         ((a) > (b) ? (a) : (b))//取较大值
```

4.2 buddy_init_memmap(struct Page *base, size_t n)

功能：初始化伙伴系统。

设计思路：

1.计算 buddy 树的叶节点数目，同时计算buddy的节点数目。根据pmm_manager定义，传入的base为可分配内存空间减去opensbi、内核、page内存消耗后的可用起始地址，而n是目前空用页数目。为了不溢出内核可管理的实际内存空间，此处采用**向下取2的幂次方**。

2.程序设计中，**将 `buddy_longest` 放在page数据的后面**，需要计算 `buddy_longest` 占用页的数目，此部分页不能作为可操纵的内存空间，即buddy_base的地址位于 `buddy_longest` 末地址。

3.更新buddy_system的全局变量，记录可操纵物理内存起始地址，操作的页的数目，目前的实际空闲页数目等。

4.初始化 buddy_longest[] 树，根节点可以操作2的14幂次方页数目，根节点看作第0级，第1级节点可以操纵2的13幂次方页数目，以此类推，知道第13级可以操纵2页，第14级可以操纵1页，即管理最小

单位。

5.在page_init函数中，将所有页的Reserved标志位设置为1，此处需要将可分配的物理页的Reserved标志位清空。

4.3 static struct Page *buddy_alloc_pages(size_t n)

功能：从伙伴系统中分配一个n页数的内存块。

设计思路：

1.将n向上取2的幂次方得到size。

2.从 `buddy_longest[]` 根节点开始查找满足size的空闲块。如果找到，直接将该节点的 `buddy_longest[]` 设置为0，表示该节点没有空闲空间，同时记录该分配的起始页的索引，通过满二叉树结构，父节点唯一对应一块连续页空间（这里复用了项目中原本为page设计的函数，比如寻找页索引、页的物理地址等）；如果没有找到符合条件的空间，通过访问左右节点一直向下搜索，直到叶节点。**最终没有可用空间时返回null**。在分配时，只对分配的实际节点的 `buddy_longest[]` 进行更新，然后向上更新，没有向下更新，因为搜索可用节点的时候是从根节点开始向下搜索。

3.完成分配后，需要回溯父节点，比较父节点的左右孩子的管理空闲块大小，以最大值更新。此处没有指明记录的是左孩子还是右孩子，需要在实际操作中，加入比较。

4.利用offset，将分配的页的Reserved标志位设置为1。

代码块

```
static struct Page *buddy_alloc_pages(size_t n) {
    if (n == 0)
        return NULL;

    unsigned size = fixsize(n);
    if (buddy_longest[0] < size)
        return NULL;  // 整棵树都不够大，直接失败

    unsigned index = 0;
    unsigned node_size = buddy_size;

    // 向下查找合适的节点
    while (node_size != size) {
        unsigned left = LEFT_LEAF(index);
        unsigned right = RIGHT_LEAF(index);

        // 优先左子树，但要确保它有足够大的空闲块
        if (buddy_longest[left] >= size)
            index = left;
        else if (buddy_longest[right] >= size)
            index = right;
        else {
```

```
            // 左右都不行 -> 当前节点虽大，但内部碎片不足
            return NULL;
        }

        node_size >>= 1; // 每深入一层块大小减半
    }

    // 找到可用块
    if (buddy_longest[index] < size)
        return NULL; // 防御性检查（防止 race 或坏状态）

    buddy_longest[index] = 0;  // 标记该块已分配

    unsigned offset = (index + 1) * node_size - buddy_size;
    struct Page *page = buddy_base + offset;

    // 回溯更新父节点
    while (index) {
        index = PARENT(index);
        buddy_longest[index] = MAX(
            buddy_longest[LEFT_LEAF(index)],
            buddy_longest[RIGHT_LEAF(index)]
        );
    }

    // 设置页标志位
    for (unsigned i = 0; i < size; i++) {
        SetPageReserved(page + i);
        ClearPageProperty(page + i);
    }

    nr_buddy_free_pages -= size;
    return page;
}

```

4.4 static void buddy_free_pages(struct Page *base, size_t n)

功能：将一个n大小的内存块释放回伙伴系统，并进行合并。

设计思路：

1根据传递的base，确定分配页起始位置在buddy_system的叶节点的位置，由此确定实际分配的节点。即从叶节点节点向上传递，遇到的第一个可分配空间为0的节点即为分配的实际节点。注意，在分

配时，只对分配的实际节点的`buddy_longest[]`进行更新，然后向上更新，没有向下更新，因为搜索可用节点的时候是从根节点开始向下搜索。

2.跟新实际分配节点的`buddy_longest[]`，为该节点应该管理的页数，此处逻辑简单，因为没有碎页。

3.向上跟新父节点的b`uddy_longest[]`，如果左右节点均为空闲块，即该节点可分配将整个左右节点的空间作为一整块分配，实现合并；如果左右节点中有节点被占用，跟新该节点值为左右节点中空闲页数较大值。

3.跟新page的标志位和引用。

代码块

```
// ======== 释放物理页 ========
static void buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    unsigned size = fixsize(n);
    unsigned offset = base - buddy_base;
    unsigned node_size = 1;
    unsigned index = offset + buddy_size - 1;

    // 找到对应节点
    for (; buddy_longest[index]; index = PARENT(index))
        node_size <<= 1;

    buddy_longest[index] = node_size;
    while (index) {
        index = PARENT(index);
        node_size <<= 1;
        unsigned left = buddy_longest[LEFT_LEAF(index)];
        unsigned right = buddy_longest[RIGHT_LEAF(index)];
        if (left + right == node_size)
            buddy_longest[index] = node_size;
        else
            buddy_longest[index] = MAX(left, right);
    }

    // 恢复 page 状态
    for (unsigned i = 0; i < size; i++) {
        set_page_ref(base+i, 0);
        ClearPageReserved(base + i);
        ClearPageProperty(base + i);
    }

    nr_buddy_free_pages += size;
```

```
33   }
```

5. 功能检查

检查分为如下几步：

1. 观察page之后的末物理位置、虚拟位置和空闲页数，和初始化buddy_longest[]树之后的末物理位置、虚拟位置和空闲页数，确保buddy_system在正确的内存位置开始。
2. 刻画初始的buddy_system结构树，观察该内存管理系统是否是以2的幂次方分配。
3. 检查分配函数和释放函数：
   a. 按顺序分配小空闲块，观察分配块的地址是否连续。再按分配顺序释放空闲块，看能否恢复buddy_system的结构，可以方便观察物理内存分配情况；
   b. 无序分配和释放不指定大小的内存块。观察分配过程是否正常，能否应对复杂的分配情况，最后能否恢复buddy_system的结构。
   c. 先分配大块，再分配小块，看能否正常利用剩余的小块空间。
   d. 连续分配两个小块再释放，看能否正常合并。
4. 检查申请的物理页数目超出可分配物理页数目的情况。
5. 检查空闲页不足的情况。

结果如下：

buddy_system的初始化情况如下

代码块

```
1   buddy_init_memmap: init_base=0xffffffffc020e2f0 (PA=0x80346000), init_n=31930
2   [buddy] adjust fixsize down to 16384
3   [buddy] skip 344 pages
4   (1409024 for buddy metadata(B)
5   buddy_init_memmap done:
6   buddy_base  = 0xffffffffc02118b0 (PA=0x8049e000)
7   tree_nodes  = 32767, buddy_size=16384
```

page之后的可用初始物理地址为 `0x80346000` ，buddy_system可分配的实际起始物理地址为 `0x8049e000` ，两个地址相减可得到为344页对应的空间大小。初始内存可用空闲页为31930，需要向下取2的幂次方，此处为16384的叶节点数目，即2的14幂次方。

初始的buddy_system的结构树如下， 可以发现目前的buddy_system树完全按照我们的预期进行物理内存管理，一共有树的深度为15，分别对应2的14次幂到2的0次幂。每一层节点管理的物理内存空间均为2的次幂，从根节点到叶节点有规律递减。

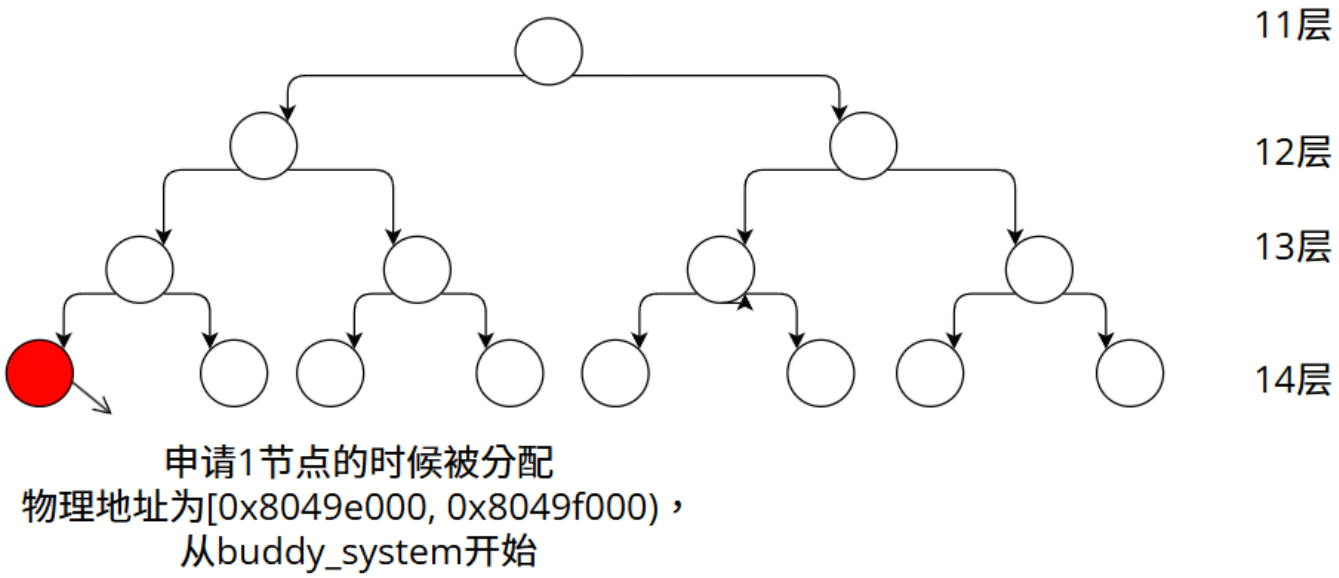
```
---- buddy tree summary ----
  Level -0 : min_free_block = 16384
  Level -1 : min_free_block = -8192
  Level -2 : min_free_block = -4096
  Level -3 : min_free_block = -2048
  Level -4 : min_free_block = -1024
  Level -5 : min_free_block = --512
  Level -6 : min_free_block = --256
  Level -7 : min_free_block = --128
  Level -8 : min_free_block = ---64
  Level -9 : min_free_block = ---32
  Level 10 : min_free_block = ---16
  Level 11 : min_free_block = ----8
  Level 12 : min_free_block = ----4
  Level 13 : min_free_block = ----2
  Level 14 : min_free_block = ----1
```

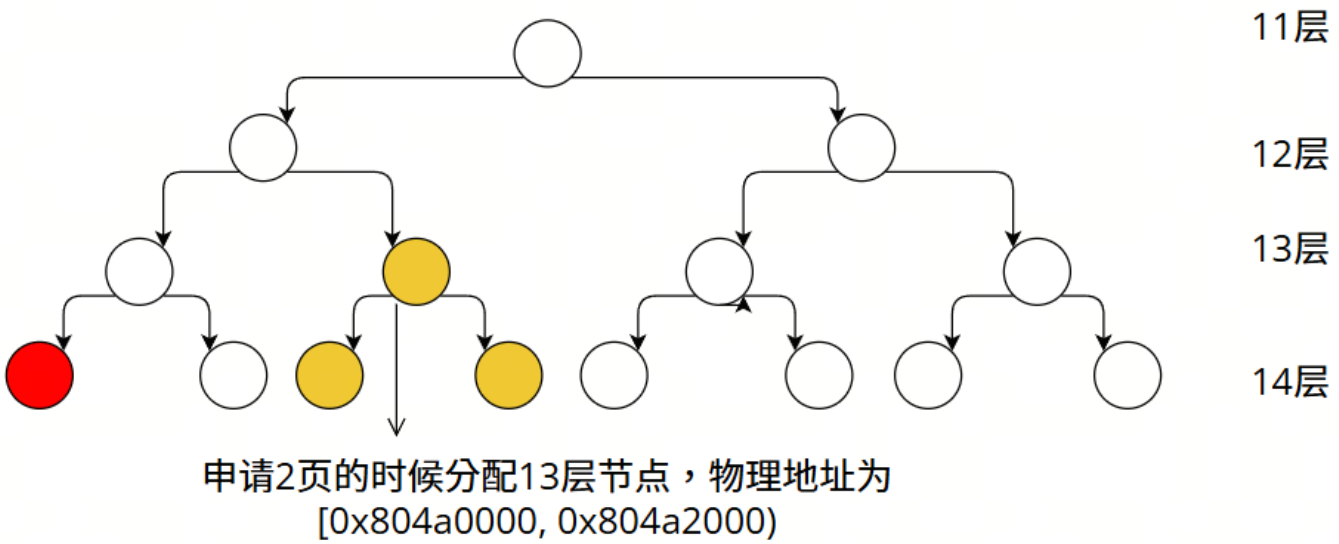
进行物理内存分配，分配过程如下

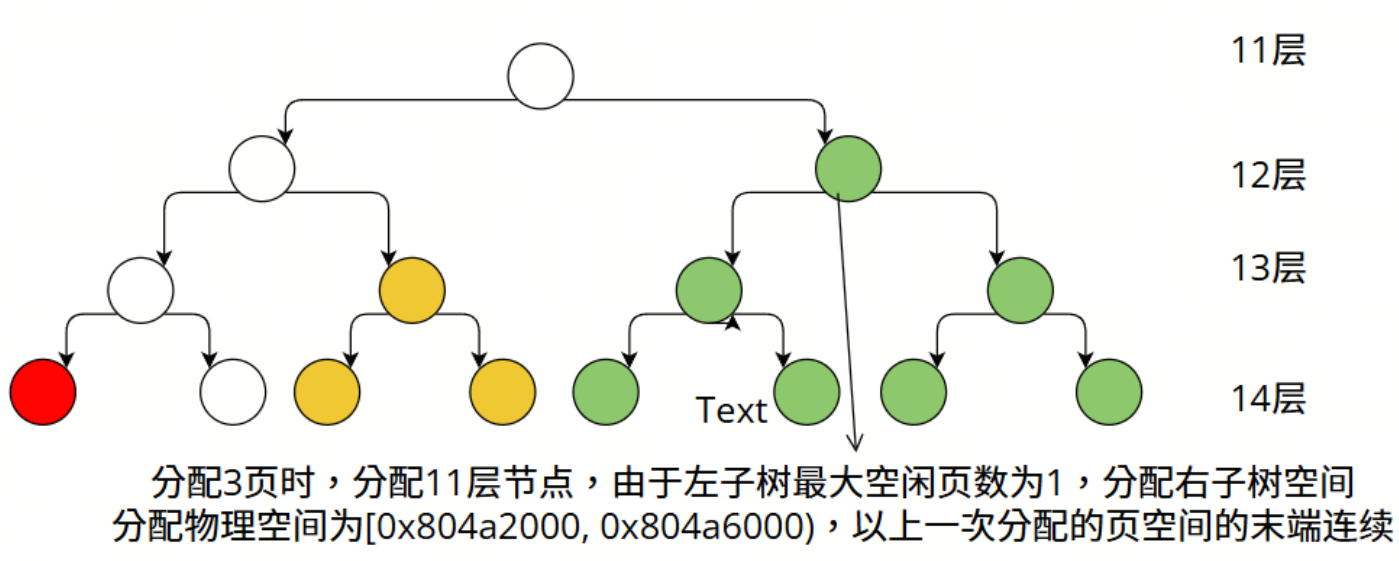
代码块

```
--- Allocation phase ---
[alloc] request 1 pages
  pages=---1  PA=[0x8049e000, 0x8049f000)
  free after alloc: 31585
[alloc] request 2 pages
  pages=---2  PA=[0x804a0000, 0x804a2000)
  free after alloc: 31583
[alloc] request 3 pages
  pages=---4  PA=[0x804a2000, 0x804a6000)
  free after alloc: 31579
[alloc] request 7 pages
  pages=---8  PA=[0x804a6000, 0x804ae000)
  free after alloc: 31571
[alloc] request 16 pages
  pages=--16  PA=[0x804ae000, 0x804be000)
  free after alloc: 31555
[alloc] request 64 pages
  pages=--64  PA=[0x804de000, 0x8051e000)
  free after alloc: 31491
```

可知当申请的页数是2的次幂时可直接申请空间，当分配的页数不足2的次幂时，需要向上取。然后我们观察物理内存是如何分配的，我们绘制了前3分分配的树状图，可以发现buddy_system严格按照我们的设计进行内存分配，优先分配当前节点左子树，能够分配连续的2的次幂空间。







分配测试完成后的buddy_system树的结构如下,每一层打印的是最小空闲块的页数，可见分配过程实现了逐渐向上更新。

代码块

```
Buddy tree after allocations:
---- buddy tree summary ----
  Level -0 : min_free_block = -8192
  Level -1 : min_free_block = -4096
  Level -2 : min_free_block = -2048
  Level -3 : min_free_block = -1024
  Level -4 : min_free_block = --512
  Level -5 : min_free_block = --256
  Level -6 : min_free_block = --128
  Level -7 : min_free_block = ---32
  Level -8 : min_free_block = ----0
  Level -9 : min_free_block = ----1
  Level 10 : min_free_block = ----0
  Level 11 : min_free_block = ----0
  Level 12 : min_free_block = ----0
  Level 13 : min_free_block = ----0
  Level 14 : min_free_block = ----0
```

超内存分配测试通过，返回null。两种情况，一种是超过内存可操控空间，一种是由于不断地分配空间，资源不足。

代码块

```
[alloc] over-allocation test:
  -> correctly failed when requesting 31501 pages
  -> correctly failed when requesting 8194 pages
```

进行物理内存释放检测。释放的页的位置即分配时的页的起始地址，释放之后总的空闲页数得到恢复,buddy_system树结构进行更新。以下展示部分释放后的buddy_system的结构变化，观察合并功能是否正常。

释放64页空间，发现level7、level8发生了更新。

代码块

```
[free] 64 pages at 0x804de000
  free after free: 31555
---- buddy tree summary ----
  Level -0 : min_free_block = -8192
  Level -1 : min_free_block = -4096
  Level -2 : min_free_block = -2048
  Level -3 : min_free_block = -1024
  Level -4 : min_free_block = --512
  Level -5 : min_free_block = --256
  Level -6 : min_free_block = --128
  Level -7 : min_free_block = ---64
```

```
  Level -8 : min_free_block = ---32
  Level -9 : min_free_block = ----1
  Level 10 : min_free_block = ----0
  Level 11 : min_free_block = ----0
  Level 12 : min_free_block = ----0
  Level 13 : min_free_block = ----0
  Level 14 : min_free_block = ----0
```

释放16页空间，发现level9、level10发生了更新。

代码块

```
[free] 16 pages at 0x804ae000
  free after free: 31571
---- buddy tree summary ----
  Level -0 : min_free_block = -8192
  Level -1 : min_free_block = -4096
  Level -2 : min_free_block = -2048
  Level -3 : min_free_block = -1024
  Level -4 : min_free_block = --512
  Level -5 : min_free_block = --256
  Level -6 : min_free_block = --128
  Level -7 : min_free_block = ---64
  Level -8 : min_free_block = ---32
  Level -9 : min_free_block = ---16
  Level 10 : min_free_block = ----1
  Level 11 : min_free_block = ----0
  Level 12 : min_free_block = ----0
  Level 13 : min_free_block = ----0
  Level 14 : min_free_block = ----0
```

释放8页空间，发现level10、level11发生了更新。

代码块

```
[free] 8 pages at 0x804a6000
  free after free: 31579
---- buddy tree summary ----
  Level -0 : min_free_block = -8192
  Level -1 : min_free_block = -4096
  Level -2 : min_free_block = -2048
  Level -3 : min_free_block = -1024
  Level -4 : min_free_block = --512
  Level -5 : min_free_block = --256
  Level -6 : min_free_block = --128
  Level -7 : min_free_block = ---64
```

```
  Level -8 : min_free_block = ---32
  Level -9 : min_free_block = ---16
  Level 10 : min_free_block = ----8
  Level 11 : min_free_block = ----1
  Level 12 : min_free_block = ----0
  Level 13 : min_free_block = ----0
  Level 14 : min_free_block = ----0
```

将分配的空间完全释放后，buddy_system还原到初始状态。

采用如下的乱序分配、释放指令，发现最终buddy_system能够复原结构，此处也进行了单步的内存跟踪，发现内存分配、释放在乱序下也正常执行。

代码块

```
    cprintf("=== buddy_check nosort ===\n\n");
    struct Page *p1 = buddy_alloc_pages(3);
    struct Page *p2 =buddy_alloc_pages(98);
    struct Page *p3 =buddy_alloc_pages(3);
    buddy_free_pages(p3,3);
    buddy_free_pages(p2,98);
    struct Page *p4 =buddy_alloc_pages(100);
    struct Page *p5 =buddy_alloc_pages(1);
    buddy_free_pages(p5,1);
    buddy_free_pages(p4,100);
    struct Page *p6 =buddy_alloc_pages(1086);
    buddy_free_pages(p1,3);
    buddy_free_pages(p6,1086);
    cprintf("after nosort allocations and frees, buddy tree:\n");
    print_buddy_summary();
```

连续分配两个小空间，再进行释放，发现能正确合并。

代码块

```
=== Merge Test ===
  pages=---2  PA=[0x8049f000, 0x804a1000)
  pages=---2  PA=[0x804a1000, 0x804a3000)
After allocations (merge test):
---- buddy tree summary ----
  Level -0 : min_free_block = -8192
  Level -1 : min_free_block = -4096
  Level -2 : min_free_block = -2048
  Level -3 : min_free_block = -1024
  Level -4 : min_free_block = --512
  Level -5 : min_free_block = --256
```

```
    Level -6 : min_free_block = --128
    Level -7 : min_free_block = ---64
    Level -8 : min_free_block = ---32
    Level -9 : min_free_block = ---16
    Level 10 : min_free_block = ----8
    Level 11 : min_free_block = ----4
    Level 12 : min_free_block = ----0
    Level 13 : min_free_block = ----0
    Level 14 : min_free_block = ----1
```

先分配一个大的物理空间，再分配小的物理空间，看物理内存管理系统能否利用碎片空间。

代码块

```
    cprintf("\n=== Split Test ===\n");
    unsigned total = nr_buddy_free_pages;
    struct Page *p_big = buddy_alloc_pages(4095);
    assert(p_big != NULL);
    struct Page *p_small1 = buddy_alloc_pages(512);
    struct Page *p_small2 = buddy_alloc_pages(512);
    struct Page *p_small3 = buddy_alloc_pages(3);
    assert(p_small1 && p_small2);

    cprintf("After allocations (split test):\n");
    print_buddy_summary();
    cprintf("free pages now: %u\n", (unsigned)nr_buddy_free_pages);

    buddy_free_pages(p_small1, 512);
    buddy_free_pages(p_small2, 512);
    buddy_free_pages(p_small3, 3);
    buddy_free_pages(p_big, 4096);
    assert(nr_buddy_free_pages == total);
```

经过测试函数，buddy_system的分配、释放、合并、边界检测和处理资源不足各功能均成功实现。

6. 总结

采用满二叉树实现buddy_system，利用满二叉树本身的结构优势，可以快速定位指定的空闲块并且进行回溯更新。释放和分配过程都是logN时间复杂度，且树深在此处只有15层。并且分配和释放都是按照2的幂方整块处理，不需要处理空闲碎片，大大简化内存管理过程。但是也可以发现，由于采用2的幂方整块分配，会造成空间的浪费。

4. 扩展练习Challenge：任意大小的内存单元Slub分配算法

1. 引言

为什么需要slub?

为什么需要slub?对于伙伴系统来说，有下面几个弊端：

1. 造成内部碎片化：以2的幂次的页数来分配块，如果频繁申请3K、5K等这样的奇数块，就会造成大量的块内空闲，导致浪费内存。
2. 造成外部碎片化：对于想要合并的空闲块，当他的伙伴正在被利用，那么他无法合并，造成外部内存碎片。随着系统长期运行，内存不断被申请与释放，不同大小的块交错排列，即使有合并机制，也常常找不到匹配的伙伴，故而空闲块被割裂、分布零散。
3. 频繁小分配导致碎片堆积：当系统频繁分配小块（几十字节或几百字节）时， 即使伙伴系统能提供页级（4KB、8KB）分配， 也会让多个小对象共享一页，难以再回收整页，从而在页级别形成外部碎片。

这就是为什么 Linux 要引入 SLUB、SLAB 这类“对象缓存层”，在伙伴系统之上做更细粒度的分配管理。

接下来，我们就仔细分析slub的实现过程：

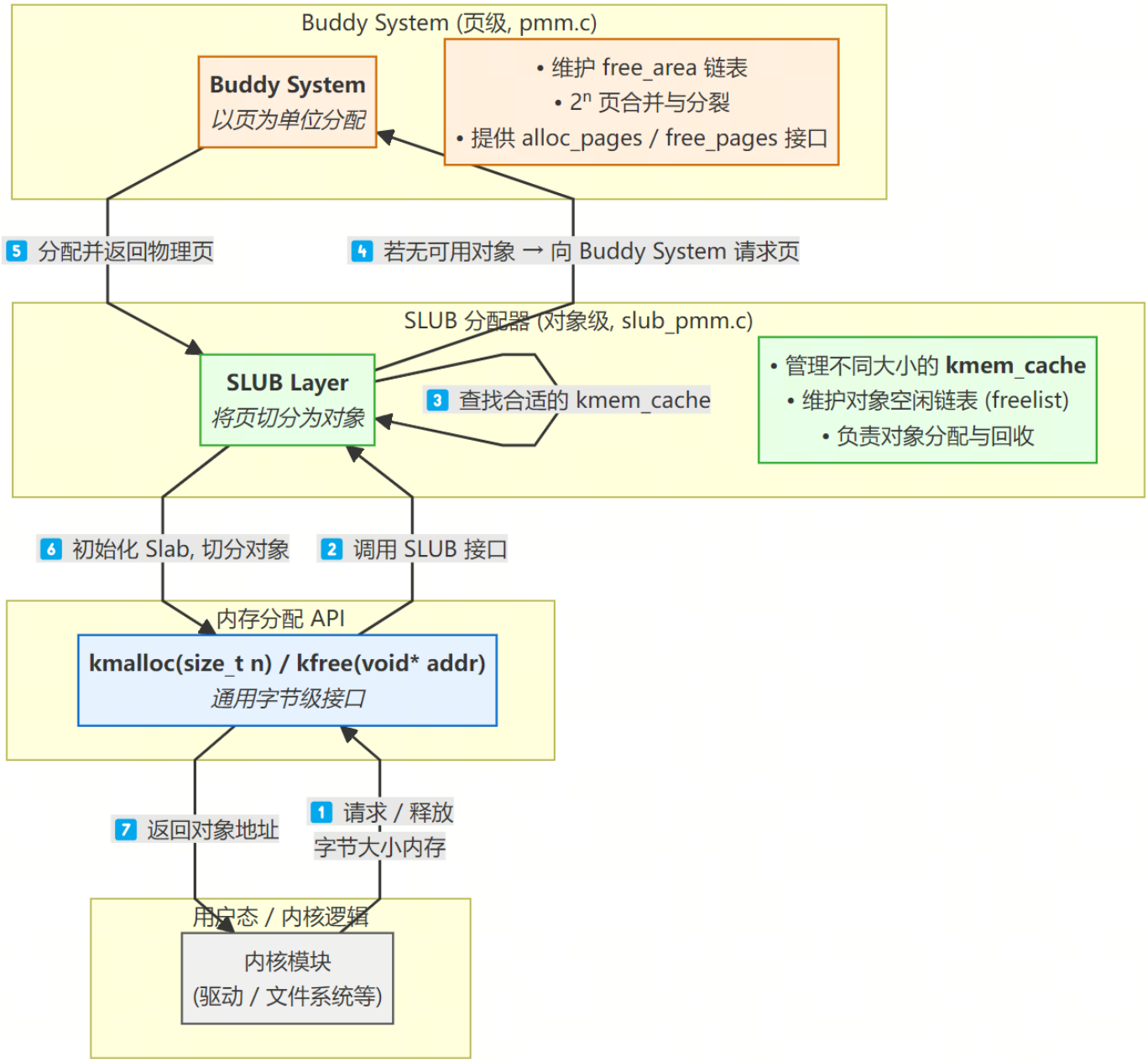
设计理念

SLUB分配器的核心思想是通过“对象缓存”的机制来解决上面提出的问题。它将将从伙伴系统批发来的整页内存（Slab），预先切分成大量固定大小的小对象（Object），零售给内核各模块。通过复用这些对象来摊销分配成本并消除内部碎片。

具体目标：

- 实现一个分层的内存管理模型。
- 为不同大小的对象建立独立的缓存池。
- 高效地复用已释放的对象和Slab。
- 对于超出SLUB处理能力的大块内存请求，能平滑地回退到底层伙伴系统。

slub的工作结构大致如下：



2.核心设计与数据结构分析

2.1. 分层内存管理架构

物理内存管理器 (PMM - Buddy System)： PMM面向以页为单位管理整个物理内存。它对上层提供 alloc_pages 和 free_pages 接口。通俗点说，PMM和SLUB就是一个批发商和零售商的关系，SLUB分配器面向内核提供以字节 为单位的 kmalloc 和 kfree 接口。它在需要时向PMM申请页面，并在内部进行精细化管理。

2.2. 核心数据结构 struct kmem_cache (位于 slub_pmm.c)

这是SLUB的灵魂，每个 kmem_cache 实例管理一种特定大小的对象。

代码块

```
1   struct kmem_cache {
2       list_entry_t slabs_full;      // 全满slab链表
```

```
    list_entry_t slabs_partial;   // 部分空闲slab链表
    list_entry_t slabs_free;      // 完全空闲slab链表

    uint32_t object_size;         // 本缓存中每个对象的大小
    uint32_t objects_per_slab;    // 本缓存中每个slab可以存放的对象数
    uint32_t slab_pages;          // 每个slab占用的物理页数量（为了简单起见，初始值设为1）

    // lock_t lock;               // 因为处理器是单核，所以先不用锁

    char name[32];                // 缓存名称
};
```

- slabs_full , slabs_partial , slabs_free (list_entry_t)：
  - slabs_partial （部分空闲）是分配时的首选，效率最高。
  - slabs_free （完全空闲）是次选，需要初始化。
  - slabs_full （已满）不参与分配，但对象被释放后其Slab可能移至 slabs_partial 。
- object_size , objects_per_slab : 描述了缓存每个对象多大，每个Slab能装多少个。

2.3. struct Page 的扩展使用 (位于 slub_pmm.c 和 pmm.h)

通过复用机制，已有的 struct Page 结构来存储Slab的元数据，避免了额外的内存开销。

在memlayout.h中，定义了结构体struct Page：

代码块

```
struct Page {
    int ref;                        // page frame's reference counter
    uint64_t flags;                 // array of flags that describe the status of the page frame
    unsigned int property;          // the num of free block, used in first fit pm manager
    list_entry_t page_link;         // free list link

    // --- for SLUB ---
    void *freelist;                 // Pointer to the first free object in the slab
    size_t inuse;                   // Number of allocated objects in the slab
    struct kmem_cache *cache;       // Points back to the cache it belongs to
};

```

在物理内存中，每个页都拥有一个对应的结构体，功能是记录该页的具体情况：这个页是空闲的、被分配的，还是被保留的？对于ref，作为其引用计数，表明有多少指针指向了这个页，当指针数量为0的时候才可以释放这个页。 如果这个页是空闲的，它需要通过 page_link 字段被挂载到 Buddy System 对应的 free_area 链表上，以便未来可以被分配。

当一个页被 alloc_pages 分配给 SLUB 分配器后，这个页的所有权就从 Buddy System 转移到了 SLUB。对于 Buddy System ，这个页现在是 "已分配" 状态。它不再关心这个页内部发生了什么，也不会再将它链接到任何 free_area 链表上。因此， struct Page 中的 page_link 字段对于 Buddy System 来说，暂时是无用的。SLUB 需要管理这个页（现在称之为 Slab），追踪这个 Slab 中哪些小对象是空闲的，以及这个Slab 属于哪个缓存池 (kmem_cache)。这就是我们的复用机制的切入点。

SLUB 分配器巧妙地利用了 struct Page 中那些在页被分配后就 "闲置" 的字段，赋予它们新的含义。

代码块

```
static void
slab_init(struct kmem_cache *cache, struct Page *page) {
    page->inuse = 0;        //设置已用对象计数为0
    page->cache = cache;        //记录自己属于哪个kmem_cache
    page->freelist = NULL;        //初始化Slab内部的空闲对象链表
//循环分割，构建freelist
    void *slab_base = (void *)(page2pa(page) +
PHYSICAL_MEMORY_OFFSET);//slab内核基地址虚拟地址的计算
    for (int i = 0; i < cache->objects_per_slab; i++) {
        void *obj = (void *)((char *)slab_base + i * cache->object_size);
        *(void **)obj = page->freelist;
        page->freelist = obj;
    }
}
```

对于空闲链表的构建，我们用循环的第一次 (i=0) 和第二次 (i=1) 来举例：

- 当 i = 0 时 :
  - obj 指向第一个对象。
  - *(void **)obj = page->freelist; // page->freelist 初始为 NULL ，所以第一个对象的起始位置被写入 NULL 。
  - page->freelist = obj; // freelist 头指针现在指向第一个对象。
  - 此时链表是： freelist -> obj_0 -> NULL
- 当 i = 1 时 :
  - obj 指向第二个对象。

        - *(void **)obj = page->freelist; // page->freelist 现在是第一个对象的地址。所以第二个对象的起始位置被写入第一个对象的地址。
        - page->freelist = obj; // freelist 头指针更新为指向第二个对象。
        - 此时链表是： freelist -> obj_1 -> obj_0 -> NULL
    - 循环结束后， freelist 就变成了一个从最后一个对象指向第一个对象的、完整的空闲链表。

代码块

```
static bool
kmem_cache_grow(struct kmem_cache *cache) {
    struct Page *page = alloc_pages(cache->slab_pages);
    if (page == NULL) {
        return false;
    }

    slab_init(cache, page);

    list_add(&(cache->slabs_free), &(page->page_link));
    return true;
}
```

当页属于 SLUB 时， page_link 用于将这个 Slab（由 struct Page 代表）链接到 kmem_cache 的 slabs_full , slabs_partial , 或 slabs_free 链表中。

3.关键函数与接口实现 (slub_pmm.c 和 slub_pmm.h)

3.1. 初始化流程： kmalloc_init()

kmalloc_caches 全局数组：它是一个 kmem_cache 指针数组，用于实现通用的 kmalloc 接口。

- 遍历从 KMALLOC_MIN_SHIFT 到 KMALLOC_MAX_SHIFT (即可申请的最小和最大字节数)的大小，为每种2的幂次大小的对象创建一个专属的 kmem_cache 实例（通过调用 kmem_cache_create ）。

3.2. 分配核心逻辑： kmalloc() -> kmem_cache_alloc()

代码块

```
void *
kmem_cache_alloc(struct kmem_cache *cache) {
    list_entry_t *le;
    struct Page *page;

    // 1. 尝试在部分空闲链表中找到一个slab
    if (!list_empty(&(cache->slabs_partial))) {
```

```
8           le = list_next(&(cache->slabs_partial));
9           page = le2page(le, page_link);
10      } else {
11          // 2. 如果部分空闲链表为空，尝试完全空闲链表
12          if (list_empty(&(cache->slabs_free))) {
13              // 3. 如果完全空闲链表也为空，扩展缓存
14              if (!kmem_cache_grow(cache)) {
15                  return NULL; // 扩展失败
16              }
17          }
18          le = list_next(&(cache->slabs_free));    //获取 slabs_free 链表中第一个元素的 list_entry_t 节点的地址。
19          page = le2page(le, page_link);        //获取对应的 struct Page 的起始地址。此时， page 指向一个完全空闲的 Slab。
20      }
21
22      // 此时，'page'指向一个有空闲对象的slab
23      void *obj = page->freelist;
24      page->freelist = *(void **)obj;
25      page->inuse++;
26
27      // 更新slab链表
28      list_del(le);
29      if (page->inuse == cache->objects_per_slab) {
30          // slab现在已满
31          list_add(&(cache->slabs_full), le);
32      } else {
33          // slab现在是部分空闲
34          list_add(&(cache->slabs_partial), le);
35      }
36
37      return obj;
38  }
39
```

kmalloc(size) 的职责：

a. 大内存回退： 检查 size 是否超出 KMALLOC_MAX_SHIFT ，如果超出，则直接调用 alloc_pages ，即利用伙伴系统。

b. 缓存定位： 调用 size_to_index(size) 找到最适合 size 的缓存仓位。

c. 委派分配： 调用 kmem_cache_alloc() 执行真正的分配。

kmem_cache_alloc(cache) 的详细步骤：

a. 首选 slabs_partial : 尝试从部分空闲链表获取一个Slab。

b. 次选 slabs_free : 如果 slabs_partial 为空，则从完全空闲链表获取一个Slab。

c. Slab扩容 kmem_cache_grow() : 如果上述两者都为空，则调用此函数，它会通过 alloc_pages(1) 向伙伴系统申请一个新页，并调用 slab_init() 将其初始化为一个新的空闲 Slab。

d. 对象获取 : 从选定的Slab的 freelist 中取出一个对象。

void *obj = page->freelist;
page->freelist = *(void **)obj;

- 它将被分配对象 obj 的内存区域强制转换为一个指向 void * 的指针（ void ** ），然后解引用。
- 其效果是，将 obj 指针的值（一个内存地址）解释为一个“指向指针的地址”，然后通过解引用操作，获取存储在该地址上的数据，这个数据就是下一个空闲对象的地址。这个技巧实现了在不引入任何额外数据结构（如单独的链表节点）的情况下，将被管理的对象本身串联成一个链表。
- 此操作完成了空闲链表头节点的出链操作，使 freelist 指向了下一个可用的空闲对象。

e. 状态更新 : 更新Slab的 inuse 计数，并根据其状态（是否变满）将其移动到 slabs_full 或 slabs_partial 链表。

如何理解feelist?

当一个对象被 kfree 释放后，它所占用的内存里的数据就失效了。SLUB 分配器利用这一点， 征用了这块空闲内存的起始部分 （通常是 4 或 8 字节，取决于系统是 32 位还是 64 位），用来存储一个指针 。这个指针指向下一个空闲的对象。这样，一个 Slab 内所有空闲的对象就通过这种方式，在内部形成了一个单向链表，我们称之为空闲链表 (freelist) 。

3.3. 释放核心逻辑： kfree() -> kmem_cache_free()

kfree(ptr) 的职责：

a. 通过 pa2page() 和 page->cache 判断该内存是来自SLUB还是直接来自伙伴系统。

b. 如果是SLUB对象（ page->cache != NULL ），则调用 kmem_cache_free() 。

c. 如果是大内存块，则直接调用 free_pages() 。

代码块

```
1
2  void kfree(void *ptr) {
3      if (ptr == NULL) {
4          return;
5      }
6      struct Page *page = pa2page((uintptr_t)ptr - PHYSICAL_MEMORY_OFFSET);
7      if (page->cache != NULL) {
8          // 这个指针是从slab缓存分配的
9          kmem_cache_free(ptr);
```

```
    } else {
        // 这个指针是直接从buddy系统分配的
        free_pages(page, page->property);
    }
}
```

代码块

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

    // 从对象的虚拟地址获取页面描述符
    struct Page *page = pa2page((uintptr_t)obj - PHYSICAL_MEMORY_OFFSET);
    struct kmem_cache *cache = page->cache;

    // 将对象添加回slab的空闲链表
    *(void **)obj = page->freelist;
    page->freelist = obj;

    bool was_full = (page->inuse == cache->objects_per_slab);
    page->inuse--;

    // 只有当slab状态改变时才移动它
    if (page->inuse == 0) {
        // 从部分空闲转换为完全空闲
        list_del(&(page->page_link));
        list_add(&(cache->slabs_free), &(page->page_link));
    } else if (was_full) {
        // 从全满转换为部分空闲
        list_del(&(page->page_link));
        list_add(&(cache->slabs_partial), &(page->page_link));
    }
    // 如果原来是部分空闲且仍然是部分空闲，则不做任何操作
}

```

kmem_cache_free(obj) 的详细步骤：

a. 定位元数据 : 通过 pa2page((uintptr_t)obj - PHYSICAL_MEMORY_OFFSET) 找到对象所在的 Page ，再通过 page->cache 找到对应的 kmem_cache 。

b. 归还对象 : 将对象重新加入Slab的 freelist 。

c. 状态更新：递减 inuse 计数，并根据其状态（是否从满变部分空闲，或从部分空闲变全空）将其移动到正确的链表。

4.check_slub()

第一部分：初始化与状态验证

1. **Cache 创建验证 (Test 1.1)**：检查 kmalloc_init是否成功创建了所有规格的kmem_cache。我们从 KMALLOC_MIN_SHIFT到KMALLOC_MAX_SHIFT循环，确保每个 cache 都不是 NULL。

2. **初始 Slab 状态验证 (Test 1.2)**：一个刚初始化的分配器应该是“干净”的。我们检查所有 kmem_cache 的 slabs_full、slabs_partial 和 slabs_free 链表是否都为空。这保证了分配器在起跑线上是正确的。

代码块

```
check_alloc_page() succeeded!
check_slub() starting: Enhanced SLUB test suite.
--- Part 1: Initialization & State Verification ---
  Test 1.1: Cache Creation Verification
    Cache[3]: size-8, objects_per_slab=512
    Cache[4]: size-16, objects_per_slab=256
    Cache[5]: size-32, objects_per_slab=128
    Cache[6]: size-64, objects_per_slab=64
    Cache[7]: size-128, objects_per_slab=32
    Cache[8]: size-256, objects_per_slab=16
    Cache[9]: size-512, objects_per_slab=8
    Cache[10]: size-1024, objects_per_slab=4
    Cache[11]: size-2048, objects_per_slab=2
  ✓ All caches created successfully.
  Test 1.2: Initial Slab State Verification
  ✓ All cache lists are initially empty.
--- Part 1 Passed ---
```

第一部分测试顺利通过。日志显示，从 size-8到size-204 的所有 kmem_cache都已成功创建，并且打印出了每个 cache 的 slab 能容纳的对象数量，计算正确。同时，所有 cache的初始状态（`slabs_full`,`slabs_partial`,`slabs_free`）都为空，符合出厂设置。

```
check_alloc_page() succeeded!
check_slub() starting: Enhanced SLUB test suite.
--- Part 1: Initialization & State Verification ---
  Test 1.1: Cache Creation Verification
    Cache[3]: size-8, objects_per_slab=512
    Cache[4]: size-16, objects_per_slab=256
    Cache[5]: size-32, objects_per_slab=128
    Cache[6]: size-64, objects_per_slab=64
    Cache[7]: size-128, objects_per_slab=32
    Cache[8]: size-256, objects_per_slab=16
    Cache[9]: size-512, objects_per_slab=8
    Cache[10]: size-1024, objects_per_slab=4
    Cache[11]: size-2048, objects_per_slab=2
  ✓ All caches created successfully.
  Test 1.2: Initial Slab State Verification
  ✓ All cache lists are initially empty.
--- Part 1 Passed ---
```

第二部分：Slab 内部对象分配与回收

这部分深入到单个 `slab` 的内部，检验核心的 `freelist` 机制。

1. **Slab 耗尽与填满测试 (Test 2.1)**：我们以 `size-64` 的 `cache` 为例，持续分配对象，直到一个 `slab` 被完全用尽。此时，我们验证该 `slab` 是否被正确地移动到了 `slabs_full` 链表。然后，我们再分配一个对象，触发新 `slab` 的创建。接着，我们从 `full` 的 `slab` 中释放一个对象，验证它是否会回到 `slabs_partial` 链表，并且这个过程不会错误地释放物理页面（因为 `slab` 应该被缓存）。

2. **对象地址检查 (Test 2.2)**：我们连续分配两个相同大小的对象，然后检查它们的虚拟地址。对于一个刚从新 `slab` 分配的对象，它们的地址应该是连续的，并且间隔恰好等于对象的大小。这验证了 `freelist` 指针移动的正确性。

代码块

```
--- Part 2: Slab Internal Object Allocation & Reclamation ---
  Test 2.1: Slab Exhaustion & Filling Test
    Testing with size-64 cache, objects_per_slab=64
    Initial free pages: 30177
    After filling one slab, free pages: 30175
    ✓ First slab is now in slabs_full list.
    After allocating N+1 object, free pages: 30174
    After freeing one object from full slab, free pages: 30174
    ✓ Slab moved from full to partial, page cached.
  Test 2.2: Object Address Pattern Check
    addr1=0xffffffffc0622000, addr2=0xffffffffc0622040, diff=64
```

```
    ✓ Object addresses follow expected pattern.
--- Part 2 Passed ---
```

第二部分测试也通过。

- **Test 2.1**：初始空闲页为 `30177`。分配 64 个 64 字节对象后（填满一个 `slab`），空闲页变为 `30175`。这里消耗了 2 个物理页，一个用于 `slab` 本身，另一个可能用于存储 `obj_array` 的元数据。`slab` 被正确移入 `slabs_full` 链表。再分配第 65 个对象，触发了新 `slab` 的创建，空闲页变为 `30174`，又消耗了 1 页。然后，释放 `full slab` 中的一个对象，空闲页数量 `30174` 保持不变，证明了 `slab` 被成功缓存并转移到了 `partial` 链表，没有被错误释放。

- **Test 2.2**：连续分配的两个 64 字节对象，地址分别为 `0x...c0622000` 和 `0x...c0622040`，地址差 `diff` 恰好是 64 字节。这精确地验证了 `freelist` 的指针操作是正确的。

```
--- Part 2: Slab Internal Object Allocation & Reclamation ---
  Test 2.1: Slab Exhaustion & Filling Test
    Testing with size-64 cache, objects_per_slab=64
    Initial free pages: 30177
    After filling one slab, free pages: 30175
    ✓ First slab is now in slabs_full list.
    After allocating N+1 object, free pages: 30174
    After freeing one object from full slab, free pages: 30174
    ✓ Slab moved from full to partial, page cached.
  Test 2.2: Object Address Pattern Check
    addr1=0xffffffffc0622000, addr2=0xffffffffc0622040, diff=64
    ✓ Object addresses follow expected pattern.
--- Part 2 Passed ---
```

第三部分：多 Slab 协同工作

当内存使用变得复杂时，`kmem_cache` 的调度逻辑就至关重要了。

1. **Partial 链表优先使用测试 (Test 3.1)**：这是 SLUB 性能的关键。我们先手动制造一个 `full slab` 和一个 `partial slab`。然后，我们再请求分配一个新对象。此时，我们期望分配器优先从 `partial slab` 中分配，而不是傻傻地去 `buddy system` 申请新页来创建 `slab`。我们通过检查分配前后 `nr_free_pages()` 的值是否变化来验证这一点。

代码块

```
--- Part 3: Multi-Slab Collaborative Work ---
  Test 3.1: Partial List Priority Test
    ✓ Allocation used partial slab, no new page allocated.
--- Part 3 Passed ---
```

第三部分，核心的调度逻辑测试通过。测试日志明确指出 `Allocation used partial slab, no new page allocated`。这说明当我们请求新对象时，分配器从 `slabs_partial` 链表中找到了可用的空间，而不是去麻烦 `buddy system`。

```
--- Part 3: Multi-Slab Collaborative Work ---
  Test 3.1: Partial List Priority Test
    ✓ Allocation used partial slab, no new page allocated.
--- Part 3 Passed ---
```

第四部分：边界与异常条件

1. **最大尺寸分配 (Test 4.1)**：测试分配器是否能处理其设计范围内的最大内存请求（`1 << KMALLOC_MAX_SHIFT`）。

2. **超大尺寸回退 (Test 4.2)**：当请求的内存大小超过 SLUB 的处理能力时，它必须能正确地甩锅给底层的 `buddy system`。我们请求一个 `PGSIZE * 2` 的大内存，并验证 `buddy system` 是否真的被调用（通过检查物理页的消耗）。

3. **内存耗尽测试 (Test 4.3)**：模拟系统物理内存被耗尽的场景。在这种极限情况下，`kmalloc` 应该返回 `NULL`，而不是导致系统崩溃。

代码块

```
--- Part 4: Boundary & Exception Conditions ---
  Test 4.1: Maximum Size Allocation
    kmalloc(2048) returned 0xffffffffc0625800
    ✓ Maximum size allocation test passed.
  Test 4.2: Oversized Allocation Fallback
    kmalloc(8192) returned 0xffffffffc0626000
    ✓ Fallback to buddy system used exactly 2 pages.
    ✓ Fallback memory correctly returned to buddy system.
  Test 4.3: Memory Exhaustion Test (Limited)
    ✓ Memory exhaustion handling works correctly.
--- Part 4 Passed ---
```

最后，边界和异常测试也全部符合预期。

- **Test 4.1**：成功分配了 `2048` 字节的最大规格对象。

- **Test 4.2**：请求 `8192` 字节（`2 * PGSIZE`）的超大内存时，分配器正确回退到 `buddy system`，并且日志显示不多不少正好使用了 2 个物理页。释放后，这 2 个页也被正确归还。

- **Test 4.3**：内存耗尽测试也成功通过，表明在极端情况下 `kmalloc` 能够安全地返回失败（返回 `NULL`），而不会让系统崩溃。

```
--- Part 4: Boundary & Exception Conditions ---
  Test 4.1: Maximum Size Allocation
    kmalloc(2048) returned 0xffffffffc0625800
    ✓ Maximum size allocation test passed.
  Test 4.2: Oversized Allocation Fallback
    kmalloc(8192) returned 0xffffffffc0626000
    ✓ Fallback to buddy system used exactly 2 pages.
    ✓ Fallback memory correctly returned to buddy system.
  Test 4.3: Memory Exhaustion Test (Limited)
    ✓ Memory exhaustion handling works correctly.
--- Part 4 Passed ---
```

5.扩展练习Challenge：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，有何办法让OS获取可用物理内存范围?

在RISC-V上，充当Bootloader的OpenSBI在启动时扫描硬件或者加载一个自身配置好的DTB，这是一个树状的数据结构，用标准格式描述了所有硬件，其中就包括一个 "memory" 节点，精确定义了可用 RAM 的起始地址和大小。OpenSBI 把这个 DTB 加载到内存中的某个位置，然后把这个 DTB 的物理地址存放在一个约定的寄存器中（在 RISC-V 上就是 `a1` 寄存器）。接着，OS 内核的入口点（`entry.S`）会立刻从 `a1` 寄存器读取这个地址，并保存起来。在 C 代码初始化阶段（`pmm_init`），内核会调用一个解析函数（`dtb_init`），这个函数会去解析 DTB，找到 "memory" 节点，并通过 `get_memory_base()` 和 `get_memory_size()` 这样的函数把内存的起始地址和大小读取出来，然后 `page_init` 函数随后就使用这个范围来初始化物理内存管理器。

整个流程大概是：

代码块

```
1   OpenSBI -> DTB -> a1 寄存器 -> entry.S -> dtb_init -> pmm_init 。
```

然而，如果无法从固件（OpenSBI）那里通过设备树 DTB获得内存布局图，OS 也可以通过下面手动探测的办法去发现可用的物理内存范围。

a. 捕获异常：OS 内核从一个它确定是安全的地址（比如它自身被加载的地址）开始，以一个固定的步长（比如一个页 4KB ）去尝试读写内存。当访问到不存在或被硬件保留的地址时，CPU 的总线会产生一个错误，这个错误最终会以异常的形式被 CPU 捕获。只要 OS 在启动早期设置好了最基本的异常向量表，能够捕获到这类“加载/存储页面错误”（Load/Store Page Fault）或类似的访问错误，它就能确定这里是内存的边界。

b. 内存回绕：对于许多简单的内存，他们的控制器没有检查（或者物理上没有连接）那根能区分 `0x80...` 和 `0x88...` 的关键地址线（A27），所以从 RAM 芯片的角度看，这两个地址是无法区分的。假设一个系统只有 128MB RAM，基地址在 `0x80000000`。这意味着有效的 RAM 地址是 `0x80000000` 到 `0x87FFFFFF`。当你尝试访问 `0x88000000`（即 128MB 之后）时，内存控制器可能会简单地忽略高位的地址线，导致你实际上又访问了 `0x80000000`。 OS 可以利用这个特性来猜内存大小。

6.实验中重要知识点

1.使用结构体成员变量指针找到结构体的位置

在实验中，Page结构体中定义了page_link，他是 list_entry 类型的链表节点，通过page_link内部指针将空闲块串联起来。在分配释放过程中，我们只能直接拿到page_link的地址，利用struct成员变量在内存空间中地址连续的特点，用le2page函数找到该结构体的首位置。

2.satp寄存器

实验在内核初始化时，设计了satp寄存器的值，主要用于控制 **页表基地址** 和 **虚拟内存启用状态。**

在lab1中寄存器 `satp` 的 `MODE` 被设置为 `Bare`，在lab2实验中，先设计一个基本的页表，然后将该页表的第三级PPN存储在satp中。

位段	名称	含义
63-60	MODE	地址翻译模式（决定分页是否启用及类型）
59-44	ASID	地址空间标识符
43-0	PPN	页表的物理页号PPN，即根页表的基地址

3.三级页表的映射和超页

实验中PTE由64位构成，信号构成如下，其中第54-10位这44位为该PTE指向的下一级页首起始地址，偏移量为0。值得注意的是，PTE中存储的是页号，没有偏移量，偏移量存储在SV39的低12位中，在实际使用中，取出PTE的PPN低12位补0即可得到下一级页的物理首地址，再从VA地址中取出9位偏移量，需要 将9位偏移量左移3位，因为在非叶节点页表中，每一个PTE是8字节，即按8自己对齐，即可对应到该级对应页对应地址。

PTE的末尾有10位信号，可以设计W\X\R来控制该级页表的下一级是否为叶子节点。

和课堂中的差别是，实验中直接将三级页表的W\X\R设置为非全0，最小的内存分配单位为1GB。

那么在sv39的一个页表项占据8字节（64位），那么页表项结构是这样的：

63-54	53-28	27-19	18-10	9-8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

4.物理内存探测

在 RISC-V 中，一般由 bootloader ，即 OpenSBI 来完成的物理内存探测。它来完成对于包括物理内存在内的各外设的扫描，将扫描结果以 DTB(Device Tree Blob) 的格式保存在物理内存中的某个地方。随后 OpenSBI 会将其地址保存在 `a1` 寄存器中。实验中在entry_init时通过读取DTB信息，获取内核可操纵内存的起始地址和大小，供PMM使用。

查阅资料发现，内存总大小等信息作为设备的关键信息，应该在硬件启动初期就由CPU获得并存储，操作系统只需要通过CPU的相关协定读取即可，这个协定就是**BIOS中断**。在x86芯片中，探测物理内存布局用的BIOS中断向量是0x15，根据ax寄存器值的不同，有三种常见的方式：0xe820，0x801和0x88。简要归纳一下不同中断方式的区别，E820h 是现代 BIOS/UEFI 用来报告物理内存布局的标准接口，每次调用返回一项 20 字节的结构（基地址 + 长度 + 类型）， 能精确描述系统中所有内存区域（可用区、保留区、ACPI、NVS 等），E801h 是在 E820 出现之前，为了突破 64MB 限制而引入的接口，它只返回两段内存信息：1MB～16MB 范围（AX/CX）；16MB～4GB 范围。88h 是最早的内存探测方式，只返回 1MB 以上、最多 64MB 的连续可用内存大小（以 KB 为单位），不能识别空洞或保留区，信息极其有限。

当无法通过硬件提供的信息获取内存时，OS可以通过challenge3提供的方式进行内存探测。

5.设备树DTB的作用

DTB是一种二进制硬件描述数据结构，用于告诉操作系统系统中的物理内存布局（可能是多段的）、外设信息（UART、定时器、中断控制器等）和硬件拓扑结构。

DTB的生成是由OpenSBI 来完成的，它来完成对于包括物理内存在内的各外设的扫描，将扫描结果以DTB的格式保存在物理内存中的某个地方，并通过寄存器 `a1` 传递给内核。

DTB 在内核启动中的流程：

1️⃣ **在 `entry.s` 的 `kern_entry` 阶段：**

- Bootloader / OpenSBI 已将 DTB 的物理地址放入寄存器 `a1`。
- 汇编阶段无法解析 DTB，只是把 `a1` 的值保存到内核全局变量 `boot_dtb` 中，等待后续使用。

2️⃣ **在 `kern_init()` 阶段：**

- 内核通过调用 `dtb_init()` 读取并解析 DTB 内容；
- 提取出物理内存的起始地址、总大小、保留区（就是不可分配的区域），以及各个外设的基地址。

3️⃣ **在 `pmm_init()` 阶段：**

- 内核根据 `dtb_init()` 的解析结果，建立物理内存管理结构（如 Buddy System）；

- 把可用内存页加入空闲链表，为 SLUB、高层分配器等提供底层页框支持。

因此：

- `DTB` 的主要任务是 “告诉内核硬件有哪些资源、哪些内存可用”；
- `kern_entry` 只是保存 DTB 的地址；
- `kern_init` 才真正解析 DTB；

7.实验中未涉的重要知识点

1.物理页换出

实验只实现了物理页的分配和释放。当请求页数超过剩余空间时，就返回null，而不是选择长期没有访问的物理页（根据脏位是否写入磁盘）换出，写入新的内容。

实验中只有对ref、reserved等信号的维护，并没有处理PTE更多的信号。

2.Linux 将物理内存分为多个 zone：

- `ZONE_DMA`：供 DMA 使用；
- `ZONE_NORMAL`：供内核使用；
- `ZONE_HIGHMEM`：高端内存；
- 还有 NUMA 的 zone。

buddy 系统通常针对每个 zone 维护独立的树结构，但实验只实现了“单区管理”，但真实系统是“多区多层 buddy”。

3.单核CPU的局限性

在一个多核系统中，多个CPU核心可以同时执行内核代码。如果两个或多个核心同时调用 kmalloc 或 kfree ，并尝试修改同一个 kmem_cache 的共享数据结构（例如，从同一个 slabs_partial 链表中获取空闲对象，或者修改链表指针），就会发生数据竞争 。这会立刻导致链表损坏、内存泄漏、或将同一个内存块分配给多个使用者等灾难性后果，最终导致系统崩溃。因此，在多核环境下，对这些共享数据结构的访问 必须 通过锁来保护，确保操作的原子性。实验代码中虽然可能预留了类似 lock_t lock; 的字段，但因为是单核环境所以并未实际使用，从而简化了实现。

4.MMU与CPU、内核的关系

在现代处理器中，MMU通常 不是独立于CPU的芯片，而是集成在CPU核心内部的一个硬件电路 。它将CPU发出的 虚拟地址 翻译成 物理地址 。CPU执行的每一条指令，只要涉及到内存访问（取指令、读写数据），发出的都是虚拟地址，这个地址会先被送到MMU进行转换。MMU通过查询页表来完成地址翻译。页表本身存储在物理内存中，由**操作系统内核**来创建和维护。CPU有一个特殊的寄存器（在RISC-V中是 satp 寄存器，在x86中是 CR3 寄存器）会指向当前活动页表的物理基地址，页表里存储的是物理页帧号（PPN），再加上一些权限位（如读、写、执行、用户/内核等）。当MMU查询页表时，它会取出物理页帧号，然后将它与虚拟地址中的页内偏移（Offset）组合起来，才能形成最终的物理地址。

在计算机刚启动，内核代码开始执行的那一刻， MMU是关闭的 。此时，CPU发出的地址就是物理地址，它看到的是一个纯粹、扁平的物理内存空间。在这个“原始”状态下，内核作为系统的最高管理者，它的责任是弄清楚物理内存有多大，哪些地址范围是可用的（读取DTB）。在可用的物理内存中，找一块地方，亲手创建好第一份页表（boot_page_table_sv39）。这份初始页表将内核自己所在的虚拟地址空间映射到它实际所在的物理地址上。否则，一旦开启MMU，CPU连下一条指令都取不到了。当页表建立好之后，内核才会把页表的物理地址加载到 satp 寄存器，然后设置一个控制位， 正式开启MMU 。