操作系统 Lab2——内存管理

2313226 肖俊涛 2312282 张津硕 2311983 余辰民

密码与网络空间安全学院

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。

设计实现过程分析

First-fit算法的思路大致如下：当需要分配内存时，从头开始遍历空闲内存块链表，找到第一个大小足够满足请求的内存块，然后从这个块中分割出所需大小的内存。

程序通过一个名为 `free_area` 的全局变量来管理所有的空闲内存。这个结构体里包含一个双向链表头 `free_list` 和一个记录总空闲页数的计数器 `nr_free`。链表中的每一个节点都代表一个连续的空闲物理内存块。

下面是各个函数的作用分析：

1. `default_init()`

```
static void
default_init(void) {
  list_init(&free_list);
  nr_free = 0;//nr_free可以理解为在这里可以使用的一个全局变量，记录可用的物理页面数
}
```

- **作用**：这个函数是物理内存管理器（PMM）的初始化函数。
- **过程**：它的任务是调用 `list_init` 函数来初始化 `free_list` 链表，把它变成一个空的双向循环链表。同时，将表示总空闲页数量的 `nr_free` 变量置为0。这说明物理内存管理模块已经准备好，但还没有任何可用的空闲内存。

2. `default_init_memmap(struct Page \*base, size_t 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;
    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);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

- **作用**：这个函数在 `page_init` 函数中被调用，用于将一段探测到的、可用的连续物理内存空间（从 `base` 开始，共 `n` 个页）添加到空闲链表中进行管理。
- **过程**：
    - 首先，它会遍历这 `n` 个物理页对应的 `Page` 结构体，清除它们的 `flags` 和 `property` 字段，并将引用计数 `ref` 置为0。
    - 然后，它将这段连续内存作为一个大的空闲块。这个块的首页（`base`）的 `property` 字段被设置为 `n`，表示这个空闲块的总页数。`SetPageProperty(base)` 会设置 `base` 的 `flags` 中的 `PG_property` 位，标志着这是一个空闲块的起始页。
    - 全局空闲页数 `nr_free` 会加上 `n`。

- 最后，这个新的空闲块会被插入到 `free_list` 中。为了方便后续的合并操作，`free_list` 是一个**按物理地址从小到大排序**的链表。因此，代码会遍历链表，找到第一个地址比 `base` 大的空闲块，然后将新块插入到它前面。

3. `default_alloc_pages(size_t n)`

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    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));
        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;
}
```

- **作用**：这是实现内存分配的核心函数。按照 First-Fit 策略，从空闲链表中查找并返回一个大小至少为 `n` 的连续物理页块。
- **过程**：
    - 函数首先检查请求的页数 `n` 是否超过了当前总的空闲页数 `nr_free`。如果超过，则直接返回 `NULL`，表示分配失败。
    - 然后，它从 `free_list` 的头部开始遍历链表（`while ((le = list_next(le)) != &free_list)`）。
    - 对于每一个空闲块，它检查其大小（`p->property`）是否大于或等于请求的大小 `n`。
    - 一旦找到第一个满足条件的空闲块（这就是 "First Fit" 的体现），就停止搜索并准备从这个块进行分配。
    - 分配时，如果找到的空闲块的大小（`page->property`）正好等于 `n`，就将整个块从 `free_list` 中移除。
    - 如果空闲块的大小**大于** `n`，则进行**分割**：
        - 前 `n` 页分配出去。

- 剩下的 `page->property - n` 页形成一个新的、更小的空闲块。这个新块的起始页是 `page + n`，它的 `property` 被更新为新的大小，并被重新插入到 `free_list` 中原来的位置。
- 最后，全局空闲页数 `nr_free` 减去 `n`，并清除被分配出去的首页的 `PG_property` 标志位，然后返回这个页的指针。

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

```
    static void
    default_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));
            p->flags = 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);
                if (base < page) {
                    list_add_before(le, &(base->page_link));
                    break;
                } else if (list_next(le) == &free_list) {
                    list_add(le, &(base->page_link));
                }
            }
        }

        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;
            }
        }

        le = list_next(&(base->page_link));
        if (le != &free_list) {
            p = le2page(le, page_link);
            if (base + base->property == p) {
                base->property += p->property;
                ClearPageProperty(p);
                list_del(&(p->page_link));
            }
```

```
        }
    }
```

- **作用**：这是实现内存释放的核心函数。它将一个不再使用的、大小为 `n` 的连续物理页块（从 `base` 开始）归还给物理内存管理器。
- **过程**：
    - 首先，它将要释放的这 `n` 个页的 `flags` 和 `ref` 都清零，并将 `base` 的 `property` 设为 `n`，标记为新的空闲块。
    - 然后，它将这个新释放的块按地址顺序插入回 `free_list` 中，逻辑与 `default_init_memmap` 类似。
    - **合并操作**：为了减少内存碎片，插入后，它会检查新释放的块是否与链表中的**前一个**或**后一个**空闲块在物理上是连续的。
        - **向前合并**：检查前一个块的末尾（`p + p->property`）是否就是当前块的开头（`base`）。如果是，就将两个块合并：前一个块的大小增加 `n`，并将当前块从链表中移除。
        - **向后合并**：检查当前块的末尾（`base + base->property`）是否就是后一个块的开头（`p`）。如果是，则将两个块合并：当前块的大小增加后一个块的大小，并把后一个块从链表中移除。

物理内存分配与释放流程总结

初始化阶段：内核启动后，首先通过 pmm_init 函数进行初始化。该过程的核心是 page_init 函数，它负责探测可用的物理内存范围，然后在内存中建立一个 Page 结构体数组，为每一个物理页创建一个对应的管理单元。所有内核已占用和用于管理的内存页会被标记为“保留”。随后，default_init_memmap 将所有剩余的可用内存作为一个大的初始空闲块，并将其加入到一个按地址排序的全局双向链表 free_list 中。

分配阶段：当内核需要内存时，default_alloc_pages 函数会执行First-Fit算法。它从 free_list 头部开始顺序查找，找到第一个大小足够满足请求的空闲块。如果该块过大，就会被分裂成两部分：一部分按需分配出去，另一部分作为新的、较小的空闲块保留在链表中。

释放阶段：当内存不再需要时，default_free_pages 函数负责回收。它将待释放的内存块重新加入到 free_list 中，并保持链表的地址有序性。为了对抗内存碎片化，该函数会主动检查新释放的块是否与链表中相邻的块在物理上也是连续的。如果是，它会自动合并这些相邻的空闲块，形成一个更大的连续空闲区域，以提高未来大块内存分配的成功率。

> 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?

- **分配效率问题**：每次分配都需要从链表头部开始线性扫描，如果链表很长，或者频繁分配和释放导致前面都是些小的内存碎片，那么分配大块内存的效率会很低。
- **内存碎片问题**：First-fit 倾向于在内存的低地址区域留下很多小碎片。虽然有合并机制，但并不能完全解决问题。
- **数据结构**：可以使用更高效的数据结构来管理空闲链表，例如**分离的空闲链表**（Segregated Free Lists），根据空闲块的大小，把它们分到不同的链表中。比如，一个链表管理大小为1-4页的块，另一个管理5-16页的块等等。这样在分配时，可以直接去对应大小的链表中查找，大大减少了搜索时间。

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

> 在完成练习一后，参考kern/mm/default_pmm.c对First Fit算法的实现，编程实现Best Fit页面分配算法，算法的时空复杂度不做要求，能通过测试即可。请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放。

Best-Fit算法的核心思想是，在分配内存时遍历所有空闲内存块，找出其中能够满足请求大小、且尺寸最小的空闲块进行分配。这样可以最大限度地保留大的连续空闲空间，以满足未来可能出现的大内存请求。

1. `best_fit_init()`

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

- **功能**：此函数用于初始化物理内存管理器。它的任务是将 `free_list` 初始化为一个空的双向循环链表，并将全局空闲页计数器 `nr_free` 置零。这是内存管理的第一步，为后续将可用物理内存纳入管理做好准备。

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

```
static void
best_fit_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        /*LAB2 EXERCISE 2: 2311983*/
        // 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
        p->flags = 0;
        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: 2311983*/
            // 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;
            } else if (list_next(le) == &free_list) {
```

```
                list_add(le, &(base->page_link));
                break;
            }
        }
    }
}
```

- **功能**：此函数接收一个连续的、未被使用的物理内存区域（以 `base` 页为起点，共 `n` 页），并将其作为一个初始的大空闲块添加到 `free_list` 中。

```
p->flags = 0;
p->property = 0;
set_page_ref(p, 0);
```

  - **实现过程**：在将这段内存纳入管理之前，我们需要确保它的元数据（`Page` 结构体）是干净的。`p->flags = 0` 和 `p->property = 0` 清除了所有旧的状态标志和大小信息。`set_page_ref(p, 0)` 将引用计数清零，表示这些页当前未被任何程序使用。

```
if (base < page) {
    list_add_before(le, &(base->page_link));
    break;
} else if (list_next(le) == &free_list) {
    list_add(le, &(base->page_link));
    break;
}
```

  - **实现过程**：为了方便后续的内存合并操作，`free_list` 必须始终保持按物理地址从小到大排序。这段代码遍历链表，找到第一个地址比新块 `base` 大的空闲块 `page`，然后将 `base` 插入到它的前面。如果遍历到链表末尾仍未找到，说明 `base` 的地址是当前最大的，就将其插入到链表尾部。`break` 确保在插入操作完成后立即退出循环，避免破坏链表结构。

3. `best_fit_alloc_pages(size_t n)`

```
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: 2311983*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 遍历空闲链表，查找满足需求的空闲页框
    // 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            page = p;
            min_size = p->property;
            // break; 找到之后并不退出
```

```
        }
    }
    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;
}
```

- **功能**：这是实现 Best-Fit 算法的核心。它从 `free_list` 中查找到一个“最适合”（即尺寸 ≥ `n` 且最小）的空闲块，并将其分配出去。

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

  - **实现过程**：这是 Best-Fit 与 First-Fit 的根本区别。代码**必须遍历整个** `free_list`，而不是找到第一个满足条件的就停止。`min_size` 初始为一个非常大的值，循环中不断寻找 `p->property >= n`（满足需求）且 `p->property < min_size`（比已找到的更优）的块。最终，`page` 指针就指向了那个大小最接近 `n` 的“最佳”空闲块。
  - **分配与分裂逻辑**：找到最佳块 `page` 后，处理逻辑与 First-Fit 类似。如果块的大小 `page->property` 恰好等于 `n`，则将其从链表删除。如果大于 `n`，则进行**分裂**：将 `page` 开始的 `n` 页分配出去，剩余的 `page->property - n` 页构成一个新的空闲块，并被重新插入回 `free_list` 中。

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

```
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));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    /*LAB2 EXERCISE 2: 2311983*/
    // 编写代码
```

```
    // 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
    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);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        /*LAB2 EXERCISE 2: 2311983*/
        // 编写代码
        // 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
        // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
        // 3、清除当前页块的属性标记，表示不再是空闲页块
        // 4、从链表中删除当前页块
        // 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
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }
    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));
        }
    }
}
```

- **功能**：负责将一块已分配的内存归还给系统，并尝试与相邻的空闲块合并，以减少内存碎片。

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

- **实现过程**：把要释放的内存块（从 `base` 开始的 `n` 页）标记为一个新的空闲块。`base->property = n` 记录了尺寸；`SetPageProperty(base)` 在其 `flags` 中设置标志位，表明是一个空闲块的起始页；`nr_free += n` 更新全局空闲页计数。

```
// 向前合并
if (p + p->property == base) { ... }
// 向后合并
if (base + base->property == p) { ... }
```

- **实现过程**：这是减少内存碎片的关键步骤。在将新释放的块插入 `free_list` 后，代码会检查它在物理地址上是否与前一个或后一个空闲块相邻。如果 `p + p->property == base` 成立，说明前一个块的末尾紧邻当前块的开头，两个块可以合并成一个更大的空闲块。向后合并的逻辑同理。合并操作通过更新前一个（或当前）块的 `property` 并从链表中删除后一个块来实现。

Best-Fit算法测试结果如下所示：


```
oslab@Edison:~/labcode/lab2$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: Entering directory '/home/oslab/labcode/lab2' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c + cc kern/driver/console.c + cc kern/driver/dtb.c + cc kern/mm/best_fit_pmm.c + cc kern/mm/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 gmake[1]: Leaving directory '/home/oslab/labcode/lab2'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=20651
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
Total Score: 25/25
```

物理内存分配与释放流程总结

- **分配过程**：当调用 `best_fit_alloc_pages(n)` 时，系统会遍历整个按地址排序的 `free_list`。在遍历过程中，它会记录下所有大小 ≥ `n` 的空闲块中，尺寸最小的那一个。遍历结束后，系统就找到了“最佳”的空闲块。如果该块大于所需空间，系统会将其分裂，把需要的 `n` 页分配出去，并将剩余部分作为新空闲块放回链表。
- **释放过程**：当调用 `best_fit_free_pages(base, n)` 时，系统首先将这 `n` 页内存标记为一个新的空闲块，并根据其物理地址将其插入回 `free_list` 的正确位置以维持链表的有序性。接着，系统会检查这个新空闲块是否能与它在链表中的前驱或后继空闲块在物理地址上连续。如果连续，则执行**合并操作**，将相邻的空闲块融合成一个更大的空闲块，从而有效地减少内存碎片。

> 并回答如下问题：
>
> - 你的 Best-Fit 算法是否有进一步的改进空间?

Best-Fit 算法的改进空间

1. **采用更高效的数据结构优化查找**：可将 free_list 从按地址排序的链表，改为按尺寸排序的平衡二叉搜索树（如红黑树）。这能将 best_fit_alloc_pages 查找最佳块的时间复杂度从 O(N) 降低到 O(logN)，实现快速定位来维持空闲块的大小。
2. **采用多级分配策略**：首先根据请求的内存大小选择一个合适的块，如果没有合适的块，则分配一个更大的块并将其拆分为多个小块，以适应不同的请求。也可以维护一个空闲链表数组，每个链表负责一个特定大小范围的内存块。分配时，只需在对应尺寸的链表中查找，从而将全局扫描优化为局部搜索，大幅提升分配效率。
3. **采用边界标记优化合并**： 通过在每个内存块的头部和尾部都存储其大小和状态（即边界标记），释放块时可通过指针计算在 O(1) 时间内直接访问物理邻居的状态，实现快速合并。

Challenge 1：buddy system（伙伴系统）分配算法

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。*

一、概述

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...

在实现伙伴系统分配算法之前，首先回顾一下伙伴系统的基本概念。伙伴系统分配算法通过将内存按2的幂进行划分，以便于高效管理内存的分配与释放。该算法利用空闲链表来维护不同大小的空闲内存块，能够快速查找合适大小的块，且在合并时能够减少外部碎片。尽管其优点明显，但也存在内部碎片的问题，尤其在请求大小不是2的幂次方时。

其实我们解决的主要问题就是分配和收回，其他我们的设计基本上和first-fit差不多，但是我们需要解决的是，我们使用什么样子的数据结构来完成内存块的分配，在参考文件中，提到了一种二叉树的数据结构的实现方法，但是实际上，该方法虽然很简便，但是有一定的缺点，就是**内存开销过于大**了。。。，所以，我们采用一种数组链表的实现方法。

除此之外，关于分配和收回的相关设计，将在接下来进行。

二、开发文档

1、设计数据结构

由于我们buddy_system的每一个存储块的大小必须是2的n次方的大小，所以我们需要重新设计新的数据结构，来存放我们对应的块，参考之前的设计，我们设计新的数据结构。

```
#define MAX_BUDDY_ORDER 15  // 支持最大块 2^15 页


static list_entry_t buddy_array[MAX_BUDDY_ORDER + 1];  // 每阶的空闲链表（双向链表）
static size_t nr_free;                                 // 当前空闲页总数
extern struct Page *pages;                             // 全局页数组
extern size_t npage;
#define page2idx(p) ((p) - pages)
#define idx2page(i) (pages + (i))
```

实际上我们需要分配的是31929个页，我们采用了首先利用数组来表示每层块（所谓每层就是指2的k次页数的块），也就是**buddy_array[k]** 管理系统中所有当前“大小为 2^k 页”的空闲块，每个空闲块利用每一个链表将所有的空闲块接起来；然后nr_free表示所有空闲页的总和。

其实实际上我们分配的是31929个页，如果初始分配的话，我们可以见到的是这一定会分配成一个大块和很多小块。

2、初始化设计

把 `[base, base+n)` 区间**以尽可能大的 2^k 对齐块**划分（对齐到页号上）；插入 `buddy_array[k]` 对应链表，使链表中的块互不重叠并且覆盖所有可用页。；更新 `nr_free` 为所有页之和；使得若 `[base, base+n)` 自身对齐且大小恰为 `2^t`，则仅在 `buddy_array[t]` 中插入一个块（理想情况）。

对某个起始页号 `addr`，要将其作为大小 `2^k` 的块，需要满足：

- `2^k ≤ n`（块不能超过剩余页数）
- `addr % 2^k == 0`（页号对齐）
  满足这两个条件的最大 `k` 就是应选的阶。

```
addr = page2idx(base); remaining = n;
while remaining > 0:
    // 找最大的 block_size = 2^k <= remaining 且 addr % block_size == 0
    k = floor(log2(largest_power_of_two <= remaining and aligned))
    add block [addr, addr + 2^k - 1] into buddy_array[k]
    nr_free += 2^k
    addr += 2^k
    remaining -= 2^k
```

3、分配机制

首先，先说明分配机制的核心：从上往下找合适的块；如果太大，就不断二分拆小；直到得到刚好能容纳请求的块。

```
/* 分配 pages：把请求上调为 2^k，然后找合适块并分裂直至到达 k */
static struct Page *buddy_system_alloc_pages(size_t requested_pages) {
    assert(requested_pages > 0);
    if (requested_pages > nr_free) return NULL;

    size_t size = round_up_pow2(requested_pages);
    unsigned int order = order_of_pow2(size);
    if (order > MAX_BUDDY_ORDER) return NULL;
```

```
    /* 找到 >= order 的第一个非空链表 */
    unsigned int i = order;
    while (i <= MAX_BUDDY_ORDER && list_empty(&buddy_array[i])) i++;
    if (i > MAX_BUDDY_ORDER) return NULL;

    /* 自上而下分裂直到 order */
    while (i > order) {
        /* split the first block in buddy_array[i] into two of order i-1 */
        buddy_system_split(i);
        i--;
    }

    /* 现在 buddy_array[order] 非空，拿出第一个块 */
    list_entry_t *le = list_next(&buddy_array[order]);
    struct Page *page = le2page(le, page_link);
    list_del(le);
    ClearPageProperty(page);

    /* 更新总空闲页 */
    nr_free -= (1u << order);
    return page;
}
```

当系统收到一个内存页分配请求时，它首先把请求页数向上取整为最接近的 2 的幂（保证块大小合法），然后确定对应的阶次 `order`。接着，它从这一阶开始查找空闲块链表 `buddy_array[order]`，若当前阶次没有可用块，就向更高阶（更大的块）寻找。当找到一个足够大的块后，如果块比需要的大，就从高阶往低阶不断将其对半分裂，每次分裂出的另一半重新放入相应阶次的空闲链表中。最终，当块大小恰好满足请求时，从对应链表中取出一个块首页、标记为已分配，并更新系统的空闲页计数 `nr_free`。

4、释放机制

首先，释放机制的核心就是：从下往上找不断找到相同的块进行合并。

```
/* 释放 pages：base 必须是块头页（property 指示阶）或者 n 给出释放页数与块一致 */
static void buddy_system_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    /* 将 n 调整为块大小（必须是 2^k），并计算order */
    size_t blk_size = round_up_pow2(n);
    unsigned int order = order_of_pow2(blk_size);
    assert((1u << order) == blk_size);

    /* 将 base 标记为该 order 的块头并插入 */
    base->property = order;
    SetPageProperty(base);
    list_add(&buddy_array[order], &base->page_link);

    /* 尝试合并 */
    struct Page *left = base;
    while (order < MAX_BUDDY_ORDER) {
        struct Page *buddy = get_buddy(left, order);
        if (buddy == NULL) break;
        /* 只有当伙伴存在且是空闲且同阶时才合并 */
```

```
        if (!PageProperty(buddy) || buddy->property != order) break;
        /* 从链表中删除 left 和 buddy（注意可能 buddy 在链表任何位置） */
        list_del(&left->page_link);
        list_del(&buddy->page_link);
        /* 计算合并后新的左侧基址（较小的地址）*/
        if (left > buddy) {
            struct Page *tmp = left;
            left = buddy;
            buddy = tmp;
        }
        /* 新块阶升一 */
        order++;
        left->property = order;
        /* 将新的大块插回对应链表以便可能继续合并 */
        list_add(&buddy_array[order], &left->page_link);
    }
```

当系统释放一个内存块时，它首先把该块标记为空闲并插入对应阶的链表，然后计算其伙伴块位置。如果发现伙伴也空闲且大小相同，就将两者从链表中删除并合并为一个更大阶的块，再把新块插入更高阶链表中继续尝试合并。这个自下而上的合并过程会持续进行，直到伙伴不存在或已被使用。最终，所有相邻同阶的空闲块都会自动拼接成更大的连续块，从而有效减少外部碎片并维持内存的层次平衡结构。

三、测试样例

关于测试，我们需要一个合理的测试案例用来表示我们的结果是正确的。ok，首先的是一个初始化的测试。

我们先展示我们的测试代码。

```
static void buddy_system_check_easy_alloc_and_free_condition(void) {
    cprintf("CHECK OUR EASY ALLOC CONDITION:\n");
    cprintf("当前总的空闲页的数量为：%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;

    cprintf("首先,p0请求16页\n");
    p0 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("然后,p1请求4页\n");
    p1 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("最后,p2请求500页\n");
    p2 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("p0 idx=%u\n", (unsigned)page2idx(p0));
    cprintf("p1 idx=%u\n", (unsigned)page2idx(p1));
    cprintf("p2 idx=%u\n", (unsigned)page2idx(p2));
```

```
    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);
    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);

    cprintf("CHECK OUR EASY FREE CONDITION:\n");
    cprintf("释放p0...\n");
    free_pages(p0, 10);
    cprintf("释放p0后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p1...\n");
    free_pages(p1, 10);
    cprintf("释放p1后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p2...\n");
    free_pages(p2, 10);
    cprintf("释放p2后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}



static void buddy_system_check_min_alloc_and_free_condition(void) {
    struct Page *p3 = alloc_pages(1);
    cprintf("分配p3之后(1页)\n");
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, 1);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}

static void buddy_system_check_max_alloc_and_free_condition(void) {
    size_t big = 1u << MAX_BUDDY_ORDER;
    if (big > npage) big = round_down_pow2(npage);
    struct Page *p3 = alloc_pages(big);
    cprintf("分配p3之后(%u页)\n", (unsigned)big);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, big);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}

static void buddy_system_check(void) {
    cprintf("BEGIN TO TEST OUR BUDDY SYSTEM!\n");
    buddy_system_check_easy_alloc_and_free_condition();
    buddy_system_check_min_alloc_and_free_condition();
    buddy_system_check_max_alloc_and_free_condition();
}
```

1、首先我们先进行的是初始化的测试。

```
CHECK OUR EASY ALLOC CONDITION:
当前总的空闲页的数量为：31929
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

我们的内存的总页数应该是31929页，在buddy_system内存分配的算法下，我们首先做的是尽可能将空闲页数结合成一个大块（也就是2的k次方页大小的块），然后剩下的块再进行分配，结果我们就得到了相应的数组链表，也就是我们的空闲页数组链表，不难发现的是，所有空闲页加起来也就是我们相应的31929页空闲页，说明我们的初始化很正确。

2、请求和释放机制测试

我们的测试代码：

```
static void buddy_system_check_easy_alloc_and_free_condition(void) {
    cprintf("CHECK OUR EASY ALLOC CONDITION:\n");
    cprintf("当前总的空闲页的数量为：%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;

    cprintf("首先,p0请求16页\n");
    p0 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("然后,p1请求4页\n");
    p1 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("最后,p2请求500页\n");
    p2 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);
```

```
    cprintf("p0 idx=%u\n", (unsigned)page2idx(p0));
    cprintf("p1 idx=%u\n", (unsigned)page2idx(p1));
    cprintf("p2 idx=%u\n", (unsigned)page2idx(p2));

    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);
    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);

    cprintf("CHECK OUR EASY FREE CONDITION:\n");
    cprintf("释放p0...\n");
    free_pages(p0, 10);
    cprintf("释放p0后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p1...\n");
    free_pages(p1, 10);
    cprintf("释放p1后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p2...\n");
    free_pages(p2, 10);
    cprintf("释放p2后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

首先我们先申请了一个16页，在初始化的链表展示中，我们发现我们有一个16页的块，所以，直接将这个块分配，结果就是这个块不再是空闲块。

如下所示，和初始化的相比。我们失去了16页的那个块，实际上也就是分配掉了。说明分配很成功。

```
首先,p0请求16页
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来我们申请一个4页的空间，由于我们没有一个正好4页的块，所以必须要进行分裂，正好我们有一个8页的空闲块，将它分裂成两个4页的空闲块，然后进行分配，就可以了。

如下所示，我们确实将那个8页的块分裂了，然后其中一块分配掉了，剩下的一块成为了空闲的。

```
然后,p1请求4页
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  2 : [4 pages, idx=840] 【地址为0xffffffffc020f340】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下里，我们申请一个500页的空间，但是，由于我们的块都是2的k次方的页，所以距离400页最近的空闲块将被分配，也就是512页的块，上次分配后的空闲块，我们发现有一个1024页的空闲块，所以我们要将这个块进行分裂，然后将其中一个512页的块进行分配。

如下图所示，我们确实将那个1024页的空闲块分裂，其中一个分配掉了。所以只剩下一个512页的空闲块。

```
最后,p2请求500页
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  2 : [4 pages, idx=840] 【地址为0xffffffffc020f340】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order  9 : [512 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】
```

```
------------------显示完成!------------------
```

接下来就是我们的收回机制。

首先收回p0也就是16页的那个块。发现，不用合并。

```
释放p0...
释放p0后,总空闲页数为:31413
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  2 : [4 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order  9 : [512 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来我们收回那个4页的块p1，结果应该会和空闲块合并成一个8页的空闲块。当然我们的总空闲块页数也变成了31417页。

```
释放p1...
释放p1后,总空闲页数为:31417
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order  9 : [512 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】
```

```
order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来我们收回那个500页的块，实际上os给他分配了512页的块，结果显而易见，就是要合并成一个1024页的空闲块。

```
释放p2...
释放p2后,总空闲页数为:31929
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

3、测试最小请求和释放

```
static void buddy_system_check_min_alloc_and_free_condition(void) {
    struct Page *p3 = alloc_pages(1);
    cprintf("分配p3之后(1页)\n");
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, 1);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

结果如下：实际就是分配了那个一页的块。

```
分配p3之后(1页)
------------------当前空闲的链表数组:------------------
order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】
```

```
order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

4、测试最大请求和释放

```
static void buddy_system_check_max_alloc_and_free_condition(void) {
    size_t big = 1u << MAX_BUDDY_ORDER;
    if (big > npage) big = round_down_pow2(npage);
    struct Page *p3 = alloc_pages(big);
    cprintf("分配p3之后(%u页)\n", (unsigned)big);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, big);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

这种显然是请求会失败，因为确实没有这么大的块。实际上最大请求是16384页这么大的块

结果如下：

```
分配p3之后(32768页)
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】
```

```
------------------显示完成!------------------
```

Challenge2：任意大小的内存单元的 slub 分配算法

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

slub算法设计思路：

- slub (Simple List of Unused Blocks) 分配器旨在优化内存管理，通过将内存划分为适合不同大小块的“slab”，并使用链表来管理这些块。该实现支持基于页的较大内存分配以及页内的小块分配。通过使用页的属性标志，slub分配器可以有效地追踪和管理内存的分配与释放，避免了碎片问题，并通过合并相邻空闲块来提高内存利用率。
- 在具体实现时，将内存分配分成两个部分，第一部分是对大内存进行分配，用于按整页分配内存，在这一部分采用first fit算法。第二部分是对小内存进行分配，用于对小于整页的内存进行分配。

结构设计：

定义了`SlubBlock`结构体，用来分配处理小块内存的分配。

```
struct SlubBlock
{
    size_t size;            // 小块的大小
    void *page;             // 指向分配的页面
    struct SlubBlock *next; // 指向下一个小块
};
```

算法实现：

分配内存:

当需要分配内存时，首先对需要分配内存的大小进行判断，如果内存大于一整页，则采用first fit 分配算法，如果小于一整页，则调用`slub_alloc_small`函数来分配内存。

```
static struct Page *slub_alloc(size_t size)
{
    size = size * PGSIZE;
    if (size >= PGSIZE)
    {
        return slub_alloc_pages((size + PGSIZE - 1) / PGSIZE); // 大于一页时，直接分配整页
    }

    // 获取小块
    void *small_block_ptr = slub_alloc_small(size);
    if (small_block_ptr)
    {
        struct SlubBlock *block = (struct SlubBlock *)small_block_ptr - 1;
        return block->page; // 返回关联的页面
    }
```

```
    return NULL; // 分配失败
}
```

首先定义一个全局变量用于存储分配小块内存后剩下的内存，如下所示。`slub_block`是小块内存链表的“头节点”，用于辅助管理小块内存，它指向整个小块内存块链表的开头。链表中的每个节点表示一个小块内存块，通过 `next` 指针链接在一起。在分配和释放小块时，`slub_alloc_small` 和 `slub_free_small` 函数都会从 `slub_block` 开始遍历，以找到合适的内存块。需要注意的是，`slub_block` 的 `size` 和 `page` 成员可是以不赋值或赋特殊值，因为它不代表真实的可用内存块，而是管理和引导链表中的其他块。

```
struct SlubBlock slub_block;
```

`slub_alloc_small` 函数实现了 SLUB 分配器中针对小块内存的分配逻辑。该函数用于从小块内存链表中找到一个足够大的块并返回给用户，如果没有足够大的块，则会调用first fit 分配算法分配一个新的页面并将其分割为小块。

```
static void *slub_alloc_small(float size)
{
    // int index = get_block_index(size);
    size_t total_size = size * PGSIZE; // 计算总大小
    struct SlubBlock *temp = &slub_block;
    while (temp->next != &slub_block)
    {
        if (temp->size >= total_size)
        {
            struct SlubBlock *block = temp->next;
            temp->next = block->next;
            return (void *)(block + 1);
        }
        else
        {
            temp = temp->next;
        }
    }
    // 没有找到匹配项
    struct Page *page = slub_alloc_pages(1); // 分配一个页
    if (page == NULL)
    {
        return NULL; // 分配失败
    }
    struct SlubBlock *current_block = (struct SlubBlock *)page; // 获取页面指针
    current_block->size = 0;                                    // 设置大小
    slub_free_small((void *)(current_block + 1), 1);
    return (void *)(current_block + 1);
}
```

释放内存：

```
static void slub_free(struct Page *ptr, size_t size)
{
    size *= PGSIZE;
    if (size >= PGSIZE)
    {
        slub_free_pages(ptr, (size + PGSIZE - 1) / PGSIZE); // 释放整页
    }
    else
    {
        slub_free_small(ptr, size); // 释放小块
    }
}
```

在释放内存时，也需要分两种情况进行讨论：

- 如果页面大于一页，则调用大页的释放函数，具体实现与`default_pmm.c`中的`default_free_pages`函数一致，在此不过多展示。
- 如果页面小于一页，则调用`slub_free_small`函数来释放内存。

`slub_free_small`首先判断传入的指向要释放的内存的指针是否有效，然后将其转换为`SlubBlock`结构体的指针，并向前偏移一个单位，获取对应块的元信息（如块的大小和页面指针）。这样做是因为`ptr`指向的是实际的数据部分，而`SlubBlock`的信息存储在其前面。然后更新当前小块中剩余内存的值，最后更新其在小块链表中的位置（小块链表实际上是按照大小排列的）。

```
static void slub_free_small(void *ptr, size_t size)
{
    if (ptr == NULL)
    {
        return;
    }
    struct SlubBlock *block = (struct SlubBlock *)ptr - 1;
    size_t total_size = size * PGSIZE;
    block->size += total_size;
    struct SlubBlock *temp = &slub_block;
    while (temp->next != &slub_block)
    {
        if (temp->size <= block->size)
        {
            struct SlubBlock *next = temp->next;
            if (next == &slub_block || next > block)
            {
                temp->next = block;
                block->next = next;
                return;
            }
            temp = temp->next;
        }
        else
        {
            temp = temp->next;
        }
```

```
    }
}
```

算法测试：

测试分成两个部分：

第一个部分是基础测试，主要用于对于大于一整页内存分配的测试，该测试与`default_pmm.c`文件中`basic_check`函数基本一致，在此不过多展示。

第二个部分是对小于一整页内存分配的测试，测试样例如下，进行了64字节、128字节、256字节小块内存的分配和释放，以及重复大量64字节小块内存的分配和释放。

```
static void
slub_check(void)
{
    cprintf("测试开始\n");
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    float size0 = 64 / PGSIZE;
    float size1 = 128 / PGSIZE;
    float size2 = 256 / PGSIZE;

    assert((p0 = slub_alloc_small(size0)) != NULL);
    cprintf("分配64字节测试通过\n");
    assert((p1 = slub_alloc_small(size1)) != NULL);
    cprintf("分配128字节测试通过\n");
    assert((p2 = slub_alloc_small(size2)) != NULL);
    cprintf("分配256字节测试通过\n");
    slub_free_small(p0, size0);
    cprintf("释放64字节测试通过\n");
    slub_free_small(p1, size1);
    cprintf("释放128字节测试通过\n");
    slub_free_small(p2, size2);
    cprintf("释放256字节测试通过\n");

    cprintf("重复64字节分配测试开始\n");
    for (int i = 0; i < 1000; i++)
    {
        void *ptr = slub_alloc_small(64 / PGSIZE);
        assert(ptr != NULL);
        slub_free_small(ptr, 64 / PGSIZE);
    }
    void *ptr[64];
    for (int i = 0; i < 64; i++)
    {
        ptr[i] = slub_alloc_small(64 / PGSIZE);
        assert(ptr[i] != NULL);
    }
    for (int i = 0; i < 64; i++)
    {
        slub_free_small(ptr[i], 64 / PGSIZE);
    }
    cprintf("重复64字节分配测试通过\n");
```

```
    struct Page *page = slub_alloc(5); // 分配 5 页
    assert(page != NULL);
    slub_free_pages(page, 5); // 释放 5 页
    cprintf("分配和释放5页测试通过\n");
    cprintf("测试结束\n");
}
```

在我的 `slub_check()` 函数里，这几部分的参数可以随时改：

```
void *a = kmalloc(64);
void *b = kmalloc(128);
void *c = kmalloc(256);
...
struct Page *pg = slub_alloc_pages(5);
```

也就是：

- ✅ 小对象分配的 **字节数（64/128/256）**
- ✅ 重复分配次数（1000 次）
- ✅ 多页分配的页数（5 页）

这三类参数都可以现场改，编译一次再 `make qemu` 演示

① 展示小对象分配

```
void *a = kmalloc(32);
void *b = kmalloc(64);
void *c = kmalloc(128);
```

输出中会看到：

```
分配32字节测试通过
分配64字节测试通过
分配128字节测试通过
```

---

② 展示重复分配重用（freelist 正常）

```
for (int i = 0; i < 5; i++) {
    void *p = kmalloc(64);
    cprintf("第%d次分配: %p\n", i, p);
    kfree(p);
}
```

输出中能看到相同地址被反复使用，这证明了 freelist 逻辑正确。

---

③ 展示多页分配（与底层 PMM 协同）

```
struct Page *pg = slub_alloc_pages(3);
cprintf("分配3页，Page结构起始地址=%p\n", pg);
slub_free_pages(pg, 3);
```

这部分调用底层页分配器，分配3页后立即释放。和小对象分配不同，这里走的是 best-fit 页管理算法。

---

④ 演示 OOM 情况

如果想展示内存不足时处理，暂时可改成：

```
for (int i = 0; i < 10000; i++) {
    void *p = kmalloc(2048);
    if (p == NULL) {
        cprintf("在第%d次分配失败，触发OOM\n", i);
        break;
    }
}
```

当页用尽时，`kmalloc` 返回 NULL，算法能正确判断 OOM

> - 上层是 SLUB 小对象分配，维护不同 size-class（如 32、64、128… 字节）的缓存；每个缓存对应多个 slab，每个 slab 是 1 页。
> - 每个 slab 内的对象用单链 freelist 管理。分配时弹出一个对象，释放时压回 freelist。
> - 当 slab 空了，我会把整页归还给底层 PMM。
> - 底层 PMM 是 best-fit 页分配算法，维护空闲页块链表，支持页合并。
>
> 所以无论分配小对象还是整页内存，系统都能高效复用。

```
zjs0914@ubuntu:~/opt/riscv/labcode/lab2(x)$ make qemu
 | |__| | |_) |  __/ | | |___) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|___/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x0000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200032 (virtual)
  etext  0xffffffffc020164e (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206488 (virtual)
Kernel executable memory footprint: 26KB
memory management: slub_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
测试开始
分配64字节测试通过
分配128字节测试通过
分配256字节测试通过
释放64字节测试通过
释放128字节测试通过
释放256字节测试通过
重复64字节分配测试开始
重复64字节分配测试通过
分配和释放5页测试通过
测试结束
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
```

```
OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    ______ ____ ______
```

```
  / __ \                / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200032 (virtual)
  etext  0xffffffffc020164e (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206488 (virtual)
Kernel executable memory footprint: 26KB
memory management: slub_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
测试开始
分配64字节测试通过
分配128字节测试通过
分配256字节测试通过
释放64字节测试通过
释放128字节测试通过
释放256字节测试通过
重复64字节分配测试开始
重复64字节分配测试通过
分配和释放5页测试通过
测试结束
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
```

Challenge3：硬件的可用物理内存范围的获取方法

> - 如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围?

1. BIOS/UEFI 提供的内存映射表

BIOS E820 方法（传统方式）

原理：

- 在系统启动时，BIOS通过INT 15h中断的E820功能提供内存映射
- BIOS会返回一个内存区域描述符数组，每个描述符包含：
  - 基地址（Base Address）
  - 区域长度（Length）
  - 区域类型（Type：可用RAM、保留、ACPI区域等）
- OS在引导阶段调用这个中断，获取完整的物理内存布局

优点： 标准化、兼容性好、信息详细

UEFI 内存映射（现代方式）

原理：

- UEFI固件通过GetMemoryMap()函数提供内存映射
- 提供更详细的内存类型分类（EfiConventionalMemory、EfiBootServicesData等）
- OS通过UEFI Boot Services获取这些信息

优点： 信息更丰富、支持更大内存空间

---

2. 设备树（Device Tree）方法

原理：

- 主要用于ARM、RISC-V等嵌入式系统
- Bootloader（如U-Boot）将硬件配置信息编译成设备树二进制文件（DTB）
- 设备树中包含memory节点，描述物理内存的起始地址和大小

```
memory@80000000 {
    device_type = "memory";
    reg = <0x80000000 0x40000000>;  // 起始地址和大小
};
```

- OS在启动时解析DTB获取内存信息

优点： 灵活、适合多样化硬件平台

---

3. ACPI（高级配置与电源接口）表

原理：

- ACPI提供SRAT（System Resource Affinity Table）等表描述系统资源
- 包含内存亲和性信息、NUMA拓扑结构
- OS通过解析ACPI表获取详细的内存配置，包括热插拔内存信息

优点： 支持高级特性（NUMA、内存热插拔）

---

4. 探测式方法（Probing）

原理：

- OS主动向不同内存地址写入测试数据并读回验证
- 从低地址开始逐步探测，记录哪些区域可读写
- 需要注意：
  - 要跳过已知的设备内存映射区域（MMIO）
  - 写入要可恢复，避免破坏重要数据
  - 效率较低，通常作为备选方案

缺点： 危险、耗时、可能破坏数据

---

5. Multiboot 信息结构

原理：

- 针对使用Multiboot规范的引导加载器（如GRUB）
- Bootloader将内存映射信息放在Multiboot信息结构中
- 包含mmap（memory map）字段，描述各个内存区域

优点： 专为OS开发设计、信息完整

---

6. 固件配置表方法

原理：

- 一些架构通过特定的固件配置表传递信息
- 例如：
  - **SMBIOS**：提供系统管理信息，包括物理内存设备信息
  - **Open Firmware**：某些系统使用的固件标准
- OS读取这些标准化的数据结构获取硬件信息

---

7. 寄存器或特殊指令查询

原理：

- 某些架构提供特殊CPU指令或寄存器来查询内存配置
- 例如某些嵌入式系统的内存控制器寄存器
- OS通过读取这些硬件接口获取内存大小

适用场景： 特定架构的嵌入式系统

在uCore lab2中，**主要方法**是通过BIOS E820获取内存布局（x86架构标准方法）

知识点整理

一、实验中的重要知识点与OS原理对应关系

1. 物理内存探测（E820内存映射）

实验中的体现：

- 通过BIOS E820中断获取物理内存布局
- 解析内存区域描述符，识别可用RAM区域

OS原理知识点：

- 系统启动过程中的硬件资源发现
- 物理地址空间的布局（RAM、ROM、MMIO区域）

理解与关系：

- **含义**：E820是x86架构下获取内存信息的标准接口，OS必须先"看见"内存才能管理
- **重要性**：这是内存管理的起点，没有这一步后续都无法进行

---

2. 物理内存管理 - 连续内存分配算法

实验中的体现：

- 实现First-Fit、Best-Fit 连续内存分配算法
- 使用空闲链表（free_list）管理物理页框
- 实现 `pmm_manager` 结构体中的分配/释放接口

OS原理知识点：

- 连续内存分配（First-Fit、Best-Fit）
- 外部碎片问题
- 内存分配策略的性能权衡

理解与关系：

- **含义**：实验中以页（Page）为单位管理，而非字节级分配
- **关系**：原理讲算法思想，实验用链表数据结构实现
- **深化理解**：通过实现才能体会算法的时间复杂度差异和碎片问题

---

3. 物理页框管理（Page结构体）

实验中的体现：

- 定义 `struct Page` 描述每个物理页框
- 包含引用计数、标志位、链表指针等字段
- 建立物理页数组来管理所有页框

OS原理知识点：

- 页框（Page Frame）的概念
- 物理内存的离散管理单元
- 页框状态管理（空闲/已分配）

理解与关系：

- **含义**：Page结构体是物理页的"身份证"，记录页的所有元信息
- **关系**：原理中抽象的"页框"，实验中用具体的数据结构实现

- `ref`字段：引用计数，支持页共享
- `flags`字段：标记页状态（保留、空闲等）
- `property`字段：用于伙伴系统等高级算法

---

4. 分页机制 - 页表结构

实验中的体现：

- 建立二级页表（Page Directory + Page Table）
- 实现页目录项（PDE）和页表项（PTE）的设置
- 理解CR3寄存器与页目录基址的关系

OS原理知识点：

- 分页机制的基本原理
- 虚拟地址到物理地址的转换过程
- 多级页表的设计

理解与关系：

- **含义**：页表是虚拟内存的核心数据结构
- **实验细节**：
  - x86使用10+10+12位的地址划分（二级页表）
  - 页表项包含物理页号和标志位（P/W/U位等）
- **差异**：
  - 原理讲4级、5级页表（现代64位系统）
  - Lab2实现简化的二级页表（32位系统）
- **关键理解**：MMU如何通过页表硬件查找完成地址转换

---

5. 虚拟地址到物理地址的映射

实验中的体现：

- 实现`get_pte()`函数：获取虚拟地址对应的页表项
- 实现`page_insert()`/`page_remove()`：建立/删除映射关系
- 处理页表项不存在时的页表创建

OS原理知识点：

- 地址转换机制
- TLB（Translation Lookaside Buffer）
- 缺页异常处理（Page Fault）

理解与关系：

- **核心操作**：`va → vpn → pte → ppn → pa`
- **实验侧重**：手动建立映射关系的过程

此外，我们还自主动手实现了

1. 伙伴系统（Buddy System）

- **原理重要性**：Linux内核的核心物理内存分配算法
- **差异**：伙伴系统能更好地减少外部碎片，但需要2的幂次大小分配

2. Slub内存分配器

- **原理重要性**：用于小对象的高效分配，减少内部碎片
- **应用场景**：内核对象缓存（如进程描述符、文件对象等）

二、OS原理中重要但实验未对应的知识点

1. 交换空间与页面置换算法

- **原理**
  - LRU、Clock、Second Chance等页面置换算法
  - Swap分区的管理
  - 工作集理论、颠簸现象
- Lab2只建立基本映射
- **知识点**：这是虚拟内存"虚拟"的真正体现——支持超过物理内存的地址空间

2. 大页（Huge Pages / Super Pages）

- **原理**
  - 减少TLB miss
  - 提高大内存应用性能（数据库、虚拟化）
- x86二级页表默认4KB页，大页需要特殊支持
- **实际应用**：Linux的HugeTLBFS、Transparent Huge Pages

3. TLB管理

- **原理**
  - TLB的工作原理和刷新策略
  - ASID（Address Space ID）机制
- **实验涉及**：有简单的TLB刷新（invlpg指令），但未深入研究
- **性能影响**：TLB miss是虚拟内存的主要性能开销

DTB（Device Tree Blob） = 设备树二进制文件，英文全称 *Device Tree Blob*。

- 是一个**描述硬件信息的数据结构**，由平台固件（如 QEMU 或 OpenSBI）在内核启动时传入；
- 取代传统的“硬编码外设寄存器地址、内存布局”的方式，让 OS 启动时能**自我识别硬件配置**。

它是 *.dtb 文件的二进制形式，由 *.dts （设备树源文件）编译而来

在 RISC-V 平台（含 QEMU Virt machine）上，**OpenSBI** 在启动 OS 时会把 DTB 地址放在寄存器 `a1`。
在汇编启动代码 `entry.S` 中通常

```
_start:
    la sp, bootstacktop      # 初始化栈
    mv a0, sp                # 栈顶指针
    mv s0, a1                # 保存 dtb 地址到 s0
    call kern_init           # 跳进 C 代码
```

此时 `s0` 或 `a1` 就指向 DTB 的物理地址。

`inline` 关键字

- **主要目的**：给编译器一个“**可内联**”的**建议**——把函数体在调用点展开，**省函数调用开销**（不进栈、不做调用/返回），可能让进一步优化（常量传播、死代码删除）发生。
- **不是强制**：编译器可根据函数大小、复杂度、优化级别、是否递归等**忽略**建议。

`extern` 关键字：

告诉编译器“这个标识符**在别的翻译单元**里有**一个定义（definition）**”。

- 典型用途：在 `.h` 里**声明**全局变量 / 在 `.c` 里**定义**一次。
- 作用：建立**外部链接（external linkage）**，让多个 `.c` 文件共享同一个全局对象/函数。