

# Lab2:物理内存和页表

## 练习1：理解first-fit 连续物理内存分配算法（思考题）

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？

### 分析函数作用

在 first-fit 算法的代码中，`list_entry_t` 和 `struct Page` 是两个关键的数据结构。`list_entry_t` 是一个双向链表节点，用于构建空闲链表。`struct Page` 代表一个内存页面，包含页面的状态信息，如是否被分配、页面的引用计数等。

接下来对 `kern/mm/default_pmm.c` 中的 `default_init`, `default_init_memmap`, `default_alloc_pages`, `default_free_pages` 等相关函数进行分析

#### 1. `default_init(void)`

作用：

这个函数用于①初始化内存管理器的空闲页面链表，准备好管理页面分配和释放。②并将空闲页面计数器重置为0，以反映当前没有可用的空闲页面。

实现：

- 初始化空闲链表 `free_list` 为一个空链表。

```
list_init(&free_list);
```

- 将全局变量 `nr_free` 设置为0，这个变量用于跟踪系统中空闲页面的总数。

```
nr_free = 0;
```

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

作用：

这个函数用于①初始化一个指定的内存区域，确保初始化的前提条件，清理页面状态并设置属性。②正确地将初始化后的页面加入到内存管理的空闲链表中，以便于后续的内存分配。

实现：

- 首先使用 `assert` 检查参数 `n` 是有效的，确保请求初始化至少一个页面。

```
assert(n > 0);
```

- 通过遍历从 `base` 开始的 `n` 个页面，确保每个页面是保留状态 (`PageReserved`)，然后将这些页面的 `flags` 和 `property` 初始化为0，并将引用计数设置为0。

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

- 将 `base` 页面的 `property` 设置为 `n`，表示这组页面的属性。调用 `SetPageProperty` 将其标记为首页，最后累加更新空闲页面计数 `nr_free`。

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

- 如果空闲链表 `free_list` 为空，则直接将初始化的页面添加到链表中，否则，遍历空闲链表寻找适当的插入位置。如果 `base` 页面地址小于当前遍历到的页面地址，则在它前面插入。而如果到达链表末尾，且没有找到合适的位置，便将其添加到链表的尾部。

```
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));
        }
    }
}
```

### 3. `default_alloc_pages(size_t n)`

作用：

这个函数用于①检查请求的页面数是否有效，并防止分配超过可用页面数。②遍历空闲页面链表，找到第一个足够大的页面进行分配。③从链表中移除已分配的页面，并处理剩余页面，确保空闲链表始终保持更新。④清除分配页面的属性并更新全局空闲页面计数。⑤最终返回指向已分配页面的指针。

实现：

- 首先使用 `assert` 检查参数 `n` 是有效的，确保请求初始化至少一个页面。

```
assert(n > 0);
```

- 检查请求的页面数量是否超过当前可用的空闲页面数量 `nr_free`，如果是，则直接返回 `NULL`，表示无法分配所请求的页面。

```
if (n > nr_free) {
    return NULL;
}
```

- 遍历 `free_list` 中的每个空闲页面，寻找第一个 `property` 属性大于或等于 `n` 的页面。如果找到这样的页面，则将其存储在 `page` 变量中。

```
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;
    }
}
```

- 如果 `page` 的 `property` 大于 `n`，说明剩余的页面可以形成另一个可用块。计算剩余页面的指针并更新其 `property`，最后将其返回到空闲链表中。

```
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));
}
```

- 更新全局变量 `nr_free`，表示当前可用的空闲页面数量，随后清除该页面的属性，标志其为头页面。

```
nr_free -= n;
ClearPageProperty(page);
```

- 最后返回分配的页面指针。

```
return page;
```

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

作用：

这个函数用于①检查和清理要释放的内存页面，确保它们是有效的和可释放的。②更新要释放页面的 `property` 属性，并标记为头页面。③更新全局空闲页面计数，确保内存管理器知道当前的空闲内存。④将释放的页面插入到空闲链表的适当位置。⑤合并与相邻空闲页面的属性，以减少内存碎片。

实现：

- 使用 `assert` 检查参数 `n` 是否有效，必须请求释放至少一个页面。

```
assert(n > 0);
```

- 遍历从 `base` 开始的 `n` 个页面。确保每个页面不是保留的且不是头页面，然后清除标志并重置其引用计数。

```

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

```

- 将 `base` 页面的 `property` 属性设置为 `n`。调用 `SetPageProperty` 将其标记为一个空闲页面的头页面，并更新全局空闲页面计数 `nr_free`。

```

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

```

- 检查空闲链表是否为空，如果是，则直接将 `base` 页面添加到链表中，否则，遍历 `free_list`，找到适当的插入位置，并将页面添加到链表。

```

if (list_empty(&free_list)) {
    list_add(&free_list, &(base->page_link));
} else {
    list_entry_t* le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page* page = le2page(le, page_link);
        if (base < page) {
            list_add_before(le, &(base->page_link));
            break;
        } else if (list_next(le) == &free_list) {
            list_add(le, &(base->page_link));
        }
    }
}

```

- 检查 `base` 页面的前一个页面是否相邻，如果相邻，则合并属性，清除 `base` 页面的属性并从链表中删除它，同时更新基址 `base`。

```

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;
    }
}

```

- 类似地，检查 `base` 页面后面的相邻页面，如果相邻则合并属性并清除后面一个页面的属性，然后从链表中删除它。

```

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));
    }
}
}

```

#### 5. default\_nr\_free\_pages(void)

作用:

这个函数用于返回当前空闲页面的数量 `nr_free`

实现:

```
return nr_free;
```

#### 6. basic\_check(void) 和 default\_check(void) 函数

`basic_check(void)` 函数通过一系列分配、释放和状态检查，验证了内存管理系统的基本功能。它确保页面分配的成功与唯一性、页面引用计数的正确性、空闲链表和可分配页面的状态、页面释放后的有效性和页面复用的有效性。该测试函数是内存管理系统中不可或缺的一部分，能够帮助开发者检查和维护内存管理逻辑的正确性。

`default_check(void)` 函数通过多个步骤验证了内存管理系统的多项功能，确保系统能够正确分配和释放页面，正确处理页面的属性及状态，保证内存复用的有效性与准确性，正确管理空闲链表与空闲页面计数。

## 物理内存分配和释放过程

`First-fit` 的思想是将内存块分为已分配和自由两种状态，并且在处理内存请求时优先返回第一个足够大的空闲内存块。接下来对 `First-fit` 算法的实现过程进行分析。

#### 1. 初始化 (default\_init)

- 使用 `list_init` 初始化 `free_list`，并将 `nr_free`（可用页面数量）初始化为0。  
`free_list` 是一个双向链表，用于管理空闲内存块

#### 2. 初始化内存映射 (default\_init\_memmap)

这个函数的作用是初始化一个给定大小的内存块，并将其添加到 `free_list` 中。

- 循环遍历从 `base` 开始的每一个页面，将页面属性初始化为0，并将引用计数设为0。
- 设置第一个页面的 `property` 为块的总数，并为其添加到 `free_list` 中。如果其后还有空闲页面，会将这些链接到 `free_list`。
- 更新 `nr_free`，以记录当前的空闲页面数。

#### 3. 分配页面 (default\_alloc\_pages)

- 查找 `free_list` 中的第一个空闲块，判断其 `property` 是否可以满足请求数量 `n`。
- 如果找到了合适的块，更新其属性，将被分配的页面的标志位更新。
- 如果该块的大小大于请求的页面数 `n`，则需调整剩余页面的 `property` 并更新 `free_list`。
- 每次分配后，更新 `nr_free`，并返回指向分配页面的指针

#### 4. 释放页面 (default\_free\_pages)

- 将页面标记为一个空闲页面的头页面，并将其加入 `free_list` 中。
- 这里将每个所释放的页面的标志位和引用计数重置，同时重置其 `property`。
- 尝试合并与相邻的空闲块，确保合并后的块的 `property` 反映出新的大小，然后更新 `free_list`。

## 5. 其他辅助功能

- 检查未分配页面数量 (`default_nr_free_pages`)

返回当前的空闲页面数量 `nr_free`。

- 基础检查 (`basic_check`) 和 (`default_check`)

这些检查函数用于测试分配和释放页面的逻辑是否正常，通过多种用例验证内存分配器的正确性。

整个内存分配的过程就是通过链表对自由内存块进行管理，保证内存的有效利用。

`First-fit` 算法十分简单有效，但是，在内存利用上可能会由于碎片化而导致效率下降。

## First-fit 算法优化：

尽管 `First-fit` 算法实现起来相对简单，但它也存在一些优化空间，以提高效率和减少内存碎片，以下是一些优化方法。

### 1. 内存块排序

在 `default_init_memmap` 函数中，内存块是按照它们被初始化的顺序添加到空闲链表中的。这可能导致在链表的末尾存在许多小的空闲块，而在链表的开始处存在较大的空闲块。对空闲链表进行排序，使得块按照地址顺序排列，可以提高首次适应算法的效率。

### 2. 使用更合适的数据结构

首次适应算法使用链表来管理空闲内存块。可以考虑使用其他数据结构，如平衡树或哈希表，以加快查找和合并空闲块的速度。

### 3. 延迟分配

如果请求的内存块大于某个阈值，可以考虑延迟分配，即不立即将内存分配给请求者，而是将其保留在空闲链表中，以便可能的后续请求可以与它合并。

### 4. 内存压缩

定期或在特定条件下（如内存使用率超过某个阈值时）执行内存压缩，将所有已分配的内存块移动到内存的一端，从而创建一个更大的连续空闲块。

### 5. 使用多个空闲链表

维护多个空闲链表，每个链表对应不同大小范围的空闲块。这样可以更快地找到合适大小的空闲块，减少搜索时间。

## 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，并回答如下问题：

- 你的 `Best-Fit` 算法是否有进一步的改进空间？

## 设计实现过程

Best-fit 算法与 First-fit 算法的实现过程与代码几乎完全相同，二者的唯一的区别在分配页面时的算法。

下面对需要补充代码的部分进行填写。

### 1. `best_fit_init_mmap(struct Page *base, size_t n)` 函数

这个函数用于初始化一个内存块。

首先遍历从 `base` 开始的 `n` 个页面，确保每个页面是保留状态 (`PageReserved`)，将每一页的标志和属性信息都设为0，同时将引用计数设为0，由此可补充代码如下：

```
for (; p != base + n; p++) {
    assert(PageReserved(p));

    /*LAB2 EXERCISE 2: 2213906*/
    // 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);
}
```

然后将 `base` 页面的 `property` 属性设置为 `n`。调用 `SetPageProperty` 将其标记为头页面，并更新全局空闲页面计数 `nr_free`

接着，将 `base` 页面插入到空闲链表 `free_list` 中，如果空闲链表为空，就直接将 `base` 添加为链表的第一个元素。如果不为空，就遍历链表，找到第一个地址大于 `base` 的页面，然后将 `base` 插入到这个页面之前。如果遍历完整个链表都没有找到这样的页面，说明 `base` 是最大的块，就将 `base` 添加到链表的末尾，据此可以补充代码如下：

```
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: 2213906*/
        // 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));
        }
    }
}
```

## 2. `best_fit_alloc_pages(size_t n)` 函数

这个函数用于分配页面，也是 `First-fit` 算法和 `Best-fit` 算法唯一不同的函数。

与 `First-fit` 算法不同的是，`Best-fit` 方法需要找到满足条件但却最小的那个空间，于是需要遍历整个页链表，并维护一个 `min_size` 保存当前找到的最小连续空闲页框数量。在遍历空闲链表的过程中，首先会检查当前页面是否符合要求，并且其空闲的连续页框数是否少于记录的最小值 `min_size`。如果当前页面满足条件，并且其连续空闲页框数更小，就将 `best_fit_page` 更新为当前页面，并 `min_size` 更新为当前页面的连续空闲页框数，由此可以补充代码如下：

```
/*LAB2 EXERCISE 2: 2213906*/
// 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; // 找到了比min_size还小的页面,更新min_size的
值
    }
}
```

## 3. `best_fit_free_pages(struct Page *base, size_t n)` 函数

这个函数用于释放页面。

在遍历完从 `base` 开始的 `n` 个页面。确保每个页面不是保留的且不是头页面，然后清除标志并重置其引用计数之后，将 `base` 页面的 `property` 属性设置为 `n`。调用 `SetPageProperty` 将其标记为一个空闲页面的头页面，并增加全局空闲页面计数 `nr_free` 的值，据此可以补充代码如下。

```
/*LAB2 EXERCISE 2: 2213906*/
// 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;
```

然后空闲链表是否为空，如果是，则直接将 `base` 页面添加到链表中，否则，遍历 `free_list`，找到适当的插入位置，并将页面添加到链表。

接下来检查 `base` 与前一个页面是否相邻，如果相邻，则合并两个页面的属性，清除 `base` 页面的属性并从链表中删除它，同时更新基址 `base`，据此可以补充代码如下：

```
list_entry_t* le = list_prev(&(base->page_link));
if (le != &free_list) {
    p = le2page(le, page_link);
    /*LAB2 EXERCISE 2: 2213906*/
    // 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;
}
}
```

代码填充完成后，可以对Best-fit算法进行测试

make qemu 结果

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
entry 0xffffffffc0200032 (virtual)
etext 0xffffffffc02019ae (virtual)
edata 0xffffffffc0206010 (virtual)
end 0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physical memory map:
  memory: 0x000000007e00000, [0x0000000080200000, 0x0000000087fffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

make grade 结果

[illegible]

## 成功通过测试

# 物理内存分配和释放过程

Best-fit 算法通过选择最小的合适块，实现了更优的内存利用率，减轻了因内存碎片化带来的问题。

## 1. 初始化 (best\_fit\_init)

- 使用 `list_init` 初始化 `free_list`，并将 `nr_free`（可用页面数量）设置为0。  
`free_list` 是一个双向链表，用于管理空闲内存块。

## 2. 初始化内存映射 (best\_fit\_init\_memmap)

这一步是初始化给定大小的内存块，并将其添加到 `free_list` 中。

- 在循环中，逐页初始化每个页面的标志和属性，将其引用计数清零。
- 设置第一个页面的 `property` 为块的总数，然后将其添加到 `free_list` 中。
- 使用遍历检查，确保正确插入到 `free_list` 中，保持链表的有序性（从低地址到高地址）。

## 3. 分配页面 (best\_fit\_alloc\_pages)

- 查找 `free_list` 中的第一个空闲块，使用 Best-fit 算法匹配请求的页面数 `n`。
- 在遍历 `free_list` 的过程中，记录下满足条件的最小连续空闲页框：  
如果找到的页面的 `property` 大于 `n`，且小于当前记录的最小值，则更新当前页面为最佳匹配。
- 一旦找到合适页面，则更新页面状态（如 `PG_reserved` 和 `PG_property`），并从 `free_list` 中删除该页面。如果剩余页面仍然有效，适当更新其 `property` 并将其重新插入到链表中。
- 如果没有找到满足条件的页面，返回NULL。

## 4. 释放页面 (best\_fit\_free\_pages)

- 重置指定页面的状态标识，将其标记为释放状态。
- 设置释放块的 `property` 为释放的页块数，并更新 `nr_free`。
- 如果 `free_list` 为空，则直接将释放的页面添加为新的空闲块，否则，遍历 `free_list` 将其插入适当位置。
- 尝试将释放的块与邻近的空闲块合并，检查前一个和后一个块是否是连续的，如果是，则合并，并更新 `property`，从链表中删除合并的块，以避免内存碎片。

## 5. 辅助函数

- `best_fit_nr_free_pages`：返回目前可用的空闲页面数量。
- `best_fit_check`：用于进行基本的检查，确保内存分配和释放逻辑的正确性。

# Best-fit算法优化

## 1. 合并相邻空闲块

不仅在释放内存时，要检查相邻空闲块合并的可能性，还可以在每次分配内存后检查是否可以合并。可以设计一个定期合并空闲块的机制，比如采用定时器或在每次分配/释放后检查。

## 2. 使用多级索引或分离空闲块链表

Best-fit 算法使用线性搜索来找到最佳块，可以考虑将空闲块按大小分类，并使用更复杂的数据结构（例如基于红黑树或其他平衡树）来存储不同大小的空闲块。这样可以加速块的搜索时间。或者使用多个链表来分别存储不同大小的空闲块，快速找到最接近请求大小的块。

## 3. 延迟合并

当一个块被释放时，先不立即合并，而是在下次分配请求时进行合并。这样可以避免频繁的小内存管理操作带来的性能开销。

#### 4. 缓存机制

采用缓存机制，例如将曾经分配和释放过的块存储在一个缓存中，以加速后续同等大小的分配请求。

#### 5. 引入其他分配策略

可以考虑引入其他分配策略，如伙伴算法：分配内存时采用二进制分配，使得内存块保持一定的对齐性，这种方法有助于减少碎片化。还有首次适应（First-Fit）、循环首次适应（Next-Fit）等算法可能在分配速度上更高效。

## 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( $2^n$ ), 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。

### 算法描述

Buddy System 是一种内存分配和管理算法，它将内存划分为大小为 2 的幂的块（blocks），用于动态管理不同大小的内存请求。其核心思想是，当内存需要分配时，将大的内存块递归地分割成更小的块，直到找到合适大小的块来满足请求；而当内存释放时，试图将释放的内存块与其“伙伴”（buddy）合并，恢复为更大的内存块。

### 基本原理

- **块划分**：内存以 2 的幂为单位进行划分，每个块有一个伙伴。伙伴之间的地址关系是特定的，即两个伙伴的起始地址只相差块大小的一半，且两个块合并后能形成一个更大的块。比如，一个大小为 1024KB 的内存块，可以被划分为两个 512KB 的伙伴。
- **伙伴查找**：假设有一个块 A，其大小为  $2^k$ 。如果 A 的起始地址为 `addr`，那么它的伙伴的起始地址为 `addr XOR 2k-1`。这个按位异或操作可以快速计算出伙伴的位置。
- **内存分配**：内存分配请求根据需要的大小向上取整到最近的 2 的幂，然后在伙伴系统中查找相应大小的空闲块。如果找到的块比所需的块大，则继续将其分割，直到满足分配要求。
- **内存释放**：当内存释放时，伙伴系统会检查相应的块是否可以与其伙伴合并，形成一个更大的空闲块。如果可以合并，则继续与更大的伙伴块合并，直到不能再合并为止。

### 基本实现

#### 算法步骤

##### 1.初始化：

系统最初拥有一块连续的内存，这块内存的大小为 2 的幂。例如，如果系统拥有 1024KB 的内存，可以将其看作大小为 1024KB 的单个块。

##### 2.内存分配：

当系统接收到一个内存请求时：

- 将请求的内存大小向上取整到 2 的幂。例如，请求 300KB 的内存，则需要分配一个 512KB 的块。

- 在可用的块中查找是否有合适大小的块可分配。
- 如果找到的块比请求的块大，则将该块递归地划分成更小的伙伴块，直到找到一个大小恰好合适的块为止。
- 分配完成后，将该块标记为已使用。

### 3.内存释放：

当释放内存时：

- 查找该块的伙伴块。
- 如果伙伴块也是空闲的，则合并这两个块，形成一个更大的块。
- 继续向上查找是否可以与更大的伙伴块合并，直到不能再合并为止。

### 4.简单示例：

假设有一个大小为 1024KB 的内存块，系统开始时内存结构如下：

```
[ 1024KB 空闲 ]
```

#### 第一步：分配 200KB 的内存请求

200KB 向上取整为 512KB，分配过程如下：

- 将 1024KB 的块分割为两个 512KB 的伙伴块。
- 从中分配一个 512KB 的块。

内存结构如下：

```
[ 512KB 已使用 | 512KB 空闲 ]
```

#### 第二步：分配 100KB 的内存请求

100KB 向上取整为 128KB，分配过程如下：

- 将空闲的 512KB 块分割为两个 256KB 的伙伴块。
- 再将一个 256KB 的伙伴块分割为两个 128KB 的伙伴块。
- 分配其中一个 128KB 的块。

内存结构如下：

```
[ 512KB 已使用 | 128KB 已使用 | 128KB 空闲 | 256KB 空闲 ]
```

#### 第三步：释放 512KB 的块

释放过程中：

- 将原来已使用的 512KB 块标记为空闲。
- 由于其伙伴块是空闲的，因此将两个 512KB 块合并为一个 1024KB 的块。

内存结构如下：

```
[ 1024KB 空闲 ]
```

#### 第四步：释放 128KB 的块

释放过程中：

- 将 128KB 的块标记为空闲。
- 检查其伙伴（另一个 128KB 的块）是否空闲。
- 合并两个 128KB 的块为 256KB，更新内存结构。

最终结构：

```
[ 512KB 已使用 | 512KB 空闲 ]
```

#### 5. 核心数据结构：

伙伴系统通常使用**二叉树**或**数组**来管理内存块，每个节点表示一个内存块。树的每一层表示不同大小的块，根节点表示最大块，叶节点表示最小块。树结构的递归性质非常适合伙伴系统的块分割和合并操作。

- **根节点 (root)**：表示整个内存区域。
- **左右孩子**：代表内存块分割出的伙伴块。

### 具体实现

#### 辅助函数

```
unsigned fix_up(unsigned num) {
    int power = 0;
    while ((1 << power) < num) {
        power++;
    }
    return power;
}
// 向下舍入到最近的2的整数次幂的幂
unsigned fix_down(unsigned num) {
    int power = 0;
    while ((1 << (power + 1)) <= num) {
        power++;
    }
    return power;
}
// 计算当前节点的左孩子节点在数组中的下标
static inline unsigned LEFT_LEAF(unsigned index) {
    return index << 1;
}
// 计算当前节点的右孩子节点在数组中的下标
static inline unsigned RIGHT_LEAF(unsigned index) {
    return (index << 1) + 1;
}
// 计算当前节点的父节点在数组中的下标
static inline unsigned PARENT(unsigned index) {
    return index >> 1;
}
// 计算 a 和 b 中的最大值
static inline unsigned MAX(unsigned a, unsigned b) {
    return (a > b) ? a : b;
}
```

```

}
// 计算某个index对应的最大longest
static inline unsigned MAX_LONGEST(unsigned index) {
    return root_size >> fix_down(index);
}
// 更新某个index对应的longest
static void update(int index) {
    int l_index = LEFT_LEAF(index);
    int r_index = RIGHT_LEAF(index);
    if (l_index < node_num) { // 存在左子树
        if (r_index < node_num) { // 存在右子树
            if (root[l_index] == root[r_index] && root[r_index] ==
MAX_LONGEST(r_index)) {
                root[index] = 2 * root[r_index]; // 将该节点的longest设为两者的和
            } else {
                root[index] = MAX(root[l_index], root[r_index]); // 将该节点的
longest设为较大值
            }
        } else { // 只存在左子树
            root[index] = root[l_index];
        }
    }
}
}

```

- `unsigned fix_up(unsigned num)`: 通过不断左移 (乘以2), 直到找到一个大于或等于 `num` 的值, 将输入的 `num` 向上舍入到最近的2的整数次幂。
- `unsigned fix_down(unsigned num)`: 通过不断左移 (乘以2) 并检查下一个值是否小于或等于 `num`, 直到找到最大的符合条件的值, 将输入的 `num` 向下舍入到最近的2的整数次幂。
- `static inline unsigned LEFT_LEAF(unsigned index)`: 通过左移操作 (`index << 1`) 来获取左子节点的索引, 来计算给定节点的左子节点在数组中的索引。
- `static inline unsigned RIGHT_LEAF(unsigned index)`: 通过左移操作加1 (`(index << 1) + 1`) 来获取右子节点的索引, 从而计算给定节点的右子节点在数组中的索引。
- `static inline unsigned MAX(unsigned a, unsigned b)`: 使用条件运算符返回较大值, 来返回两个无符号整数 `a` 和 `b` 中的较大值。
- `static inline unsigned MAX_LONGEST(unsigned index)`: 通过右移操作, 计算得到的值是根节点能提供的最大块大小除以当前节点的大小, 来计算给定索引对应的子树能提供的最大块大小 (最长可用块)。
- `static void update(int index)`: 通过左子节点和右子节点的索引找到它们的值, 如果左右子节点的值相等且等于它们对应的最大块大小, 则将当前节点的值设置为左右子节点值的和; 否则, 设置为较大值, 目的是更新二叉树中某个节点的“最长可用块”值 (longest)。

### buddy\_init 函数

```

// 在init时将free_page_num设为零
static void buddy_init(void)
{
    free_page_num = 0; // 初始化空闲页数
    memset(root, 0, sizeof(unsigned) * node_num); // 清空根节点数组
}

```

**功能:** 初始化空闲页数并重置伙伴系统的根节点数组。

## buddy\_init\_memmap 函数

```
static void buddy_init_memmap(struct Page *base, size_t n)
{
    // 确保参数合法
    assert(n > 0);
    for (struct Page *p = base; p != base + n; p++) {
        assert(PageReserved(p)); // 判断当前页面是否是系统保留的
        // 清空当前页框的标志信息，并将页框的引用计数设置为0，将页面设为空闲态
        p->flags = 0;
        set_page_ref(p, 0);
        SetPageProperty(p);
    }

    // 构建二叉树存储不同节点的使用情况
    // 实际分配的大小，向上变为最近的2的整数次幂
    root_size = 1 << fix_up(n);
    // 二叉树数组的大小，从1开始编码，所以需要2倍的空间
    node_num = 2 * root_size;
    // 利用最开始的若干页建立二叉树组
    root = (unsigned *)KADDR(page2pa(base));
    // 得到二叉树组需要的页数
    int heap_page_size = node_num * sizeof(unsigned *);
    heap_page_num = heap_page_size / 4096 + (int)(heap_page_size % 4096 != 0);
    // 得到能分配的页的首地址
    base_page = base + heap_page_num;
    // 更新空闲页
    free_page_num += n - heap_page_num;

    // 由于申请的页数向向上变为最近的2的整数次幂，而系统输入的n为物理内存能容纳的最大页数，所以
    // 多申请的页数是非法的
    // 将多申请的部分的最大可分配页数设为0
    for (int i = root_size + free_page_num; i <= node_num; i++) {
        root[i] = 0;
    }
    // 将叶子节点的longest设置为1
    for (int i = root_size + free_page_num - 1; i >= root_size; i--) {
        root[i] = 1;
    }
    // 更新每个块的longest
    for (int i = root_size - 1; i > 0; i--) {
        update(i);
    }
}
```

**功能:** 初始化内存映射表，创建并配置伙伴系统的内存结构。该函数主要用于根据系统的内存布局构建二叉树结构，确保系统可以高效地进行内存管理和分配。

### 参数合法性检查:

- 使用 `assert(n > 0)` 确保 `n` 大于 0，确保传入的页数是有效的。

### 初始化页面属性:

- 遍历传入的页面范围 `base` 到 `base + n`，使用 `PageReserved(p)` 判断页面是否为保留态。



- 对于每个页面，重置标志位 `p->flags = 0`，设置引用计数为 0，并标记为可分配状态 `SetPageProperty(p)`。

#### 构建二叉树存储：

- `root_size` 被设置为 `n` 向上调整到最近的 2 的整数次幂，即调用 `fix_up(n)`。这决定了二叉树的根节点大小。
- `node_num` 表示二叉树的总节点数，是 `root_size` 的两倍，因为二叉树从 1 开始编码。

#### 分配二叉树存储空间：

- 使用最前面的若干页作为二叉树的存储空间，根节点数组 `root` 被指向这些页，转换虚拟地址使用 `KADDR(page2pa(base))`。
- `heap_page_size` 计算二叉树数组占用的总字节数，并通过页大小（通常 4096 字节）计算所需的页数 `heap_page_num`。

#### 更新空闲页数：

- 更新空闲页数 `free_page_num`，将分配给二叉树存储的页数从总页数中减去。

#### 处理越界页：

- 为防止越界，多申请的页（由于向上取整的操作）被标记为不可分配，即将这些越界节点的 `root[i]` 设置为 0。

#### 初始化叶子节点：

- 对于叶子节点，表示每个块可分配的最大大小，全部设置为 1（表示叶子块最小的单位分配）。

#### 更新二叉树的块信息：

- 从底层节点开始逐步更新树中每个父节点，使用 `update(i)` 函数更新每个节点的 `longest` 值（代表以该节点为根的最大可用块大小）。

#### `buddy_alloc_pages` 函数

```
static struct Page *buddy_alloc_pages(size_t n) {
    assert(n > 0);           // 确保请求的页面数大于0
    // 向上调整 n 为最近的2的整数次幂
    n = 1 << fix_up(n);
    // 检查是否有足够的空间
    if (n > root[1]) // 从根节点开始检查
        return NULL;
    unsigned index = 1; // 从根节点开始遍历
    size_t node_size = root_size;
    // 深度优先遍历二叉树以找到适合的块
    while (node_size != n) {
        if (root[LEFT_LEAF(index)] >= n) { // 如果左子树足够大，向左子树申请
            index = LEFT_LEAF(index);
        } else { // 否则，向右子树申请
            index = RIGHT_LEAF(index);
        }
        node_size /= 2; // 更新当前节点的大小
    }
    // 计算对应的起始页面
    int left_brother_num = index - (1 << fix_down(index)); // 计算左兄弟节点数
    struct Page *base = base_page + (left_brother_num * n); // 计算开始的页面地址
}
```



```

// 更新每个页面的状态
for (struct Page *page = base; page != base + n; page++) {
    ClearPageProperty(page); // 清除页面的属性标志
}

// 更新堆中对应节点的longest
root[index] = 0; // 将找到的块取出分配
free_page_num -= n; // 更新空闲页数

// 向上回溯至根节点，修改沿途节点的大小
while (PARENT(index)) {
    index = PARENT(index);
    update(index); // 更新每个父节点的longest
}

return base; // 返回分配的页面的起始地址
}

```

**功能：**函数根据请求的页面数 `n`，找到最合适的内存块并分配内存，同时更新伙伴系统的状态。主要流程包括页面数调整、二叉树遍历、内存块分配和状态更新。

**二叉树遍历：**从根节点开始深度优先遍历，寻找大小合适的块。左子树优先，右子树作为次选。

**状态更新：**找到合适的块后，更新其对应的二叉树节点为0，标记为已分配，清除页面属性，减少空闲页数。

**回溯更新：**通过二叉树逐层回溯，更新父节点，确保伙伴系统的状态保持一致。

**返回分配结果：**返回分配的内存块的起始地址，供调用方使用。

#### buddy\_free\_pages 函数

```

static void buddy_free_pages(struct Page *base, size_t n)
{
    assert(n > 0); // 保证n大于零
    n = 1 << fix_up(n); // 将n向上变为最近的2的整数次幂

    // 检查对应page是否是保留态且已分配，将ref设为0
    for (struct Page *p = base; p < base + n; p++)
    {
        assert(!PageReserved(p) && !PageProperty(p));
        set_page_ref(p, 0);
    }

    // 对应叶节点索引
    unsigned index = root_size + (base - base_page);

    // 找到对应节点
    for (unsigned node_size = 1; node_size != n; node_size <=< 1) {
        index = PARENT(index);
        assert(index); // 防止index为0
    }

    root[index] = n; // 将对应节点的longest设置为n
    free_page_num += n; // 更新空闲页数

    // 回溯直到根节点，更改沿途值
}

```

```

while (PARENT(index)) {
    index = PARENT(index);
    update(index);
}

```

**功能：** `buddy_free_pages` 函数将已分配的页面块 `base` 释放回内存池，并且通过回溯更新二叉树节点，使得伙伴系统保持一致性。

**块大小调整：** 释放的页面数 `n` 被调整为2的幂次，以匹配伙伴系统的内存管理规则。

**节点状态检查：** 确保释放的页面是已分配且非保留的，避免释放错误的内存块。

**二叉树更新：** 释放的页面块在伙伴系统的二叉树中找到相应的节点，并回溯更新父节点，使整个内存管理系统保持一致性。

**空闲页数更新：** 在释放内存块时，系统的空闲页数 `free_page_num` 也随之更新，确保系统状态正确。

### `basic_check()` 函数

```

static void
basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    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);
    for(int i = 0; i < MAX_ORDER; i++) {
        list_init(&(free_list(i)));
        assert(list_empty(&(free_list(i))));
    }

    for(int i = 0; i < MAX_ORDER; i++) {
        list_init(&(free_list(i)));
        assert(list_empty(&(free_list(i))));
    }
    for(int i = 0; i < MAX_ORDER; i++) nr_free(i) = 0;

    assert(alloc_page() == NULL);

    free_page(p0);
    free_page(p1);
    free_page(p2);
    assert(buddy_system_nr_free_pages() == 3);

    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(alloc_page() == NULL);
}

```

```

    free_page(p0);
    for(int i = 0; i < 0; i++) assert(!list_empty(&(free_list(i))));

    struct Page *p;
    assert((p = alloc_page()) == p0);
    assert(alloc_page() == NULL);

    assert(buddy_system_nr_free_pages() == 0);

    free_page(p);
    free_page(p1);
    free_page(p2);
}
static void
buddy_system_check(void) {}

```

该函数主要是对伙伴系统中的页面分配与释放逻辑进行基本的单元测试，以确保内存管理机制正常工作。

#### 1. 页面分配测试：

- `p0`, `p1`, `p2` 三个指针分别通过 `alloc_page()` 分配页面，使用 `assert` 确保每次分配的页面不为空。
- 使用 `assert(p0 != p1 && p0 != p2 && p1 != p2)` 来验证三次分配得到的页面是不同的。

#### 2. 页面引用计数检查：

- 使用 `assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0)` 来检查刚分配的页面引用计数是否为 0。

#### 3. 页面物理地址检查：

- `assert(page2pa(p0) < npage * PGSIZE)` 这类断言检查每个页面的物理地址是否在有效范围内。

#### 4. 空闲列表初始化与检查：

- 循环遍历 `MAX_ORDER` 级别的伙伴系统，使用 `list_init(&(free_list(i)))` 初始化每一层的空闲列表，并通过 `assert(list_empty(&(free_list(i))))` 确保每个列表在初始化后是空的。

#### 5. 页面超量分配测试：

- 在页面 `p0`, `p1`, `p2` 已经分配后，再调用 `alloc_page()`，并通过 `assert(alloc_page() == NULL)` 来验证没有多余的页面可以分配。

#### 6. 页面释放测试：

- 使用 `free_page(p0)`, `free_page(p1)`, `free_page(p2)` 来释放之前分配的三个页面。
- 然后通过 `assert(buddy_system_nr_free_pages() == 3)` 来验证释放后系统的空闲页面数量是否正确增加到 3。

## 7. 再次分配页面:

- 再次调用 `alloc_page()` 分配页面, 并通过 `assert` 确保可以成功分配三个页面。
- 验证如果超量分配, 系统会返回 `NULL`。

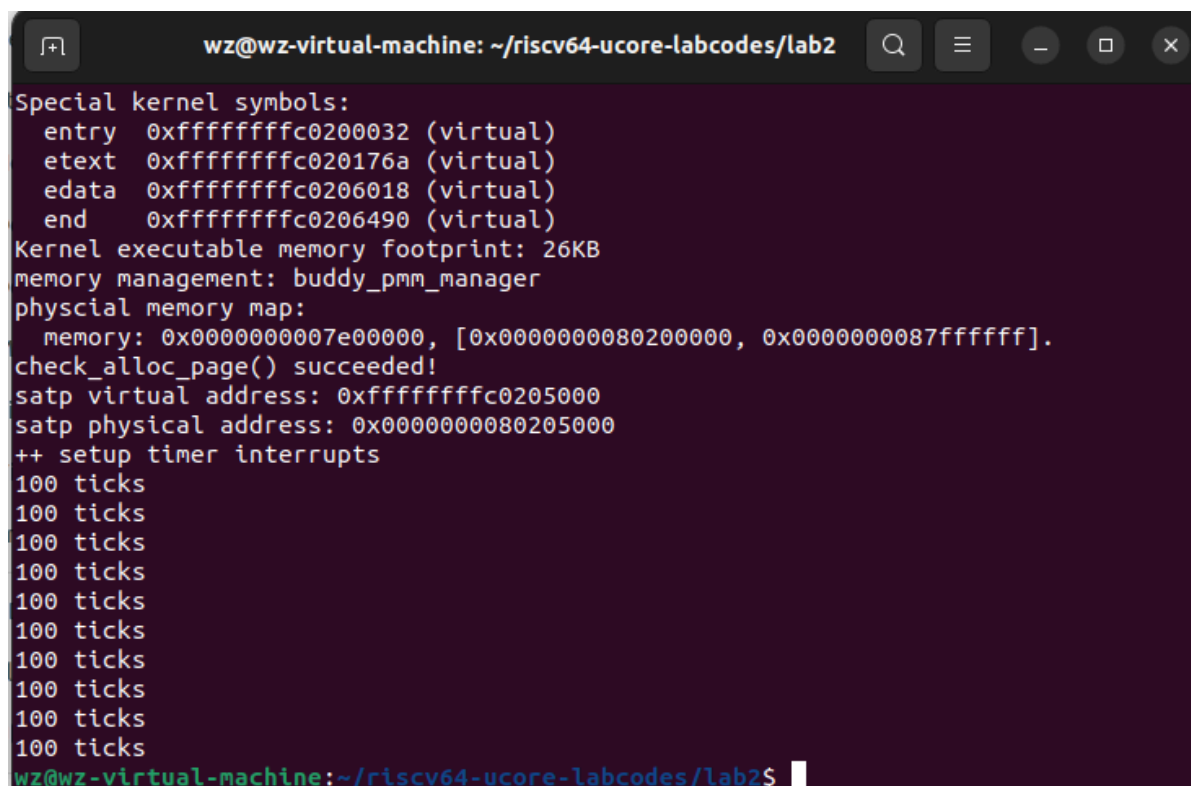
## 8. 部分页面释放与检查:

- 释放页面 `p0` 后, 检查对应的空闲列表是否为空 `assert(!list_empty(&(free_list(i))))`。
- 再次分配页面, 并确保分配到的页面与之前的 `p0` 相同。

## 9. 释放所有页面并检查空闲页面数量:

- 释放所有页面后, 调用 `buddy_system_nr_free_pages()` 验证系统中是否有正确数量的空闲页面。

执行 `make qemu`



```
wz@wz-virtual-machine: ~/riscv64-ucore-labcodes/lab2
Special kernel symbols:
  entry 0xffffffffc0200032 (virtual)
  etext 0xffffffffc020176a (virtual)
  edata 0xffffffffc0206018 (virtual)
  end    0xffffffffc0206490 (virtual)
Kernel executable memory footprint: 26KB
memory management: buddy_pmm_manager
physical memory map:
  memory: 0x000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
wz@wz-virtual-machine:~/riscv64-ucore-labcodes/lab2$
```

我们看到,使用buddy\_pmm\_mamager成功

## 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

## 解答:

首先设计 SLUB 的结构, 然后实现内存分配和释放的函数, 最后编写测试用例。

# 设计文档

## SLUB 分配器概述

SLUB (Simplified List Unordered Blocks) 是 Linux 内核中的一种高效内存分配器。其设计理念是通过减少内存碎片和提高内存使用效率来加速内存分配和释放操作。SLUB 使用了一个页作为基本分配单元，并在此基础上实现了小块内存的管理。

简单来说，slub分配算法是一种任意大小内存分配的算法，主要用于给内核分配小空间的内存。对于较大的块，可以直接使用原本的页面分配算法，对于小于一页的块，需要将一页的内容划分为小的空间分配，对于每一个分配出去的空间，我们称为slob块，在其头部存在着一个slob块基本信息的结构，用于管理。

## 架构

1. **页分配**：以页为单位分配内存。
2. **小块管理**：在每个页内维护一个自由链表，用于管理小块内存的分配和释放。

这段代码实现了一个简单的物理内存管理器，使用 SLUB (Simple List of Unused Blocks) 分配器管理物理内存页。SLUB 分配器的目标是高效分配和回收内存块，同时减少内存碎片。以下是对代码的详细分析：

## 1. SLAB 和 kmem\_cache 结构体

代码中定义了两个主要结构体：

### a. struct slab

这个结构体用于管理一页大小的内存块 (SLAB)，每个 SLAB 包含多个可分配的小块对象，具体字段如下：

- `slab_list`: 链表节点，用于将多个 SLAB 连接成链表。
- `free_count`: 当前 SLAB 中空闲对象的数量。
- `free_ptr`: 指向下一个可分配的空闲对象。
- `start_addr`: 当前 SLAB 的起始地址，通常用于标记这个 SLAB 在物理内存中的位置。

### b. struct kmem\_cache

`kmem_cache` 是一个用于管理特定大小内存块的缓存，每个缓存维护了多个 SLAB，用于按需分配和回收相同大小的内存块。它包含以下字段：

- `size`: 缓存块的大小。
- `align`: 缓存块的对齐大小（可以用于优化内存对齐）。
- `slabs_full`: 已经完全分配的 SLAB 的链表。
- `slabs_partial`: 部分空闲的 SLAB 的链表。
- `slabs_free`: 完全空闲的 SLAB 的链表。

## 2. 核心函数

### a. `kmem_cache_alloc(struct kmem_cache *cache)`

```
// 从 cache 中分配一个对象
void *kmem_cache_alloc(struct kmem_cache *cache) {
    struct slab *slab = NULL;

    // 检查部分空闲的 slab 链表
    if (!list_empty(&cache->slabs_partial)) {
        slab = le2slab(list_next(&cache->slabs_partial), slab_list);
    }
    // 如果部分空闲的 slab 不存在, 尝试从全部空闲的 slab 中分配
    else if (!list_empty(&cache->slabs_free)) {
        slab = le2slab(list_next(&cache->slabs_free), slab_list);
        list_del(&slab->slab_list);          // 将该 slab 从 slabs_free 链表中移除
        list_add(&cache->slabs_partial, &slab->slab_list); // 添加到
slabs_partial
    }
    // 如果没有可用 slab, 则分配新的 slab
    else {
        struct Page *new_page = slub_alloc_pages(1); // 分配一个新页
        if (new_page == NULL) {
            return NULL; // 分配失败
        }

        // 初始化新 slab
        slab = (struct slab *)page2kva(new_page); // 获取页面的虚拟地址
        slab->start_addr = slab;
        slab->free_ptr = slab + sizeof(struct slab); // 对象起始位置
        slab->free_count = (SLAB_SIZE - sizeof(struct slab)) / cache->size;
        list_add(&cache->slabs_partial, &slab->slab_list); // 添加到
slabs_partial
    }

    // 分配对象并更新 slab 状态
    void *obj = slab->free_ptr;
    slab->free_ptr = (char *)slab->free_ptr + cache->size;
    slab->free_count--;

    // 如果 slab 已满, 移动到 slabs_full 链表
    if (slab->free_count == 0) {
        list_del(&slab->slab_list);
        list_add(&cache->slabs_full, &slab->slab_list);
    }
    return obj;
}
```

- 首先从 `slabs_partial` 或 `slabs_free` 链表中查找一个可用的 slab
- 若不存在可用的 slab, 分配一个新页面并初始化新的 `slab` 结构
- 分配一个对象, 并将 `free_ptr` 指向下一个可用空间, 减少 `free_count`
- 若 `free_count` 为 0, 表示 slab 已满, 将 slab 移动到 `slabs_full` 链表

## b. `kmem_cache_free(struct kmem_cache *cache, void *obj)`

```
// 释放一个对象并归还到 cache
void kmem_cache_free(struct kmem_cache *cache, void *obj) {
    struct slab *slab = (struct slab *)((uintptr_t)obj & ~(SLAB_SIZE - 1)); //
    获取对象所属的 slab
    slab->free_count++;

    // 如果 slab 原来是满的，将其从 slabs_full 移动到 slabs_partial
    if (slab->free_count == 1) {
        list_del(&slab->slab_list);
        list_add(&cache->slabs_partial, &slab->slab_list);
    }

    // 如果 slab 现在完全空闲，将其从 slabs_partial 移动到 slabs_free
    if (slab->free_count == (SLAB_SIZE - sizeof(struct slab)) / cache->size) {
        list_del(&slab->slab_list);
        list_add(&cache->slabs_free, &slab->slab_list);
    }
}
```

- 通过对象地址获取所属的 slab
- 增加 `free_count` 表示释放了一个对象
- 若 slab 变为部分空闲或完全空闲，将 slab 移动到合适的链表中，以便后续分配复用

## 3. SLUB 分配器的初始化

```
// 初始化 SLUB 分配器
void slub_init(void) {
    static struct kmem_cache cache; // 声明一个 kmem_cache 类型的变量

    // 初始化 kmem_cache 的链表
    list_init(&cache.slabs_full);
    list_init(&cache.slabs_partial);
    list_init(&cache.slabs_free);

    // 初始化内存映射
    struct Page *base = alloc_pages(MAX_PAGES); // 使用适当的页面分配函数
    if (base == NULL) {
        cprintf("SLUB: Failed to allocate memory for initialization.\n");
        return;
    }
    slub_init_memmap(base, MAX_PAGES); // 初始化内存映射

    cprintf("SLUB: Allocator initialized with %d pages.\n", MAX_PAGES);
}
```

- 初始化 `kmem_cache` 中的链表：分别初始化 `slabs_full`、`slabs_partial` 和 `slabs_free` 链表，以便在分配器运行时可以追踪不同状态的 slab。
- 初始化其他相关变量或数据结构，以便 `kmem_cache` 中可以存放不同大小的缓存块。
- 调用 `slub_init_memmap` 函数，初始化实际的内存映射，标记内存中的页面是否被预留或空闲。

## 4. 内存映射初始化

```
void slub_init_memmap(struct Page *base, size_t n) {
    for (size_t i = 0; i < n; i++) {
        SetPageReserved(base + i);
    }
    cprintf("SLUB memory map initialized.\n");
}
```

这个函数负责初始化物理内存的映射。在内核启动时，需要将所有物理内存页（`Page` 结构体）标记为已保留（通过 `SetPageReserved`）。这确保了物理内存的分配与回收能够正常工作。

## 5. 页面分配与释放

### a. `slub_alloc_pages(size_t num_pages)`

这个函数用于分配连续的物理页。通过调用 `alloc_pages` 函数来实际分配物理页，并打印分配结果。如果分配失败，会返回 `NULL`。

### b. `slub_free_pages(struct Page *page, size_t num_pages)`

这个函数用于释放之前分配的物理页。通过 `free_pages` 来回收物理内存，并打印释放结果。

## 6. 空闲页统计

```
size_t slub_nr_free_pages(void) {
    size_t free_pages_count = nr_free_pages();
    cprintf("SLUB: Number of free pages: %zu.\n", free_pages_count);
    return free_pages_count;
}
```

该函数返回系统中当前的空闲物理页数量。它使用 `nr_free_pages` 函数来获取这个数值，并将结果打印出来。

## 7. SLUB 测试函数

```
void slub_check(void) {
    cprintf("SLUB: Starting self-check...\n");

    // 1. 记录当前空闲页面数量
    size_t free_pages_start = slub_nr_free_pages();
    cprintf("SLUB: Initial free pages: %zu\n", free_pages_start);

    // 2. 分配一页并检查
    struct Page *page = slub_alloc_pages(1);
    assert(page != NULL);
    cprintf("SLUB: Successfully allocated one page at %p\n", page);

    // 3. 检查分配后空闲页面计数是否减少
    size_t free_pages_after_alloc = slub_nr_free_pages();
    assert(free_pages_after_alloc == free_pages_start - 1);
    cprintf("SLUB: Free pages after allocation: %zu\n", free_pages_after_alloc);
}
```



```

// 4. 释放分配的页面并检查
slub_free_pages(page, 1);
size_t free_pages_after_free = slub_nr_free_pages();
assert(free_pages_after_free == free_pages_start);
cprintf("SLUB: Free pages after freeing: %zu\n", free_pages_after_free);

// 5. 检查分配超过最大页数的错误处理
struct Page *oversized_alloc = slub_alloc_pages(free_pages_start + 1);
assert(oversized_alloc == NULL);
cprintf("SLUB: Oversized allocation correctly returned NULL.\n");

cprintf("SLUB check complete and passed.\n");
}

```

**分配与释放测试：** `slub_alloc_pages` 和 `slub_free_pages` 函数通过直接测试单个页面的分配与释放流程，验证了分配器的基本功能。

**空闲页面计数：** 通过检查空闲页面计数的变化，验证了分配器对页面数量的精确控制。

**错误处理：** 检查超额分配是否返回 NULL，确保分配器在分配失败时能正确地进行错误处理，防止非法分配。

## 8. pmm\_manager 结构体

```

const struct pmm_manager slub_pmm_manager = {
    .name = "slub_pmm_manager",
    .init = slub_init,
    .init_memmap = slub_init_memmap,
    .alloc_pages = slub_alloc_pages,
    .free_pages = slub_free_pages,
    .nr_free_pages = slub_nr_free_pages,
    .check = slub_check,
};

```

这是一个 `pmm_manager` 结构体，用于将 SLUB 分配器的功能集成到整个系统的物理内存管理中。各个函数指针指向相应的实现函数，用于初始化、分配、释放、统计空闲页等操作。

## 执行结果

```

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 0xfffffffffc0200032 (virtual)
  etext 0xfffffffffc0201064 (virtual)
  edata 0xfffffffffc0205010 (virtual)
  end    0xfffffffffc0205458 (virtual)
Kernel executable memory footprint: 22KB
memory management: slub_pmm_manager
SLUB allocator initialized.
physical memory map:
  memory: 0x0000000007e00000, [0x00000000080200000, 0x00000000087fffffff].
SLUB memory map initialized.
SLUB check complete.
check_alloc_page() succeeded!
satp virtual address: 0xfffffffffc0204000
satp physical address: 0x00000000080204000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks

```

### 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，请问你有什么办法让 OS 获取可用物理内存范围？

**解答：**

## 1. 使用 BIOS 或 UEFI 提供的内存映射

- **BIOS (Basic Input/Output System)**：在传统 x86 架构中，操作系统可以通过调用 BIOS 的 `INT 0x15, E820h` 中断来获取系统的物理内存布局。这种方法提供了一张内存区域的映射表，描述了哪些区域可用（可供操作系统使用）以及哪些区域是保留的（例如设备内存、BIOS）。
- **UEFI (Unified Extensible Firmware Interface)**：在 UEFI 系统上，操作系统可以通过 UEFI 固件提供的 `GetMemoryMap` 函数获取系统内存映射表。这个函数会返回内存区域的类型（如可用内存、保留内存、设备专用内存等），帮助操作系统了解哪些区域可以用作物理内存管理。

示例：

- BIOS 中断 `INT 0x15, E820h` 调用返回一个内存区域表，每个表项描述一段内存的起始地址、大小以及类型。
- UEFI 提供的 `GetMemoryMap` 会返回一个 `EFI_MEMORY_DESCRIPTOR` 数组，包含不同类型的内存区。

## 2. ACPI (Advanced Configuration and Power Interface) 表

ACPI 是一种现代的标准，通过 `ACPI SRAT (System Resource Affinity Table)` 或 `ACPI SLIT (System Locality Information Table)`，操作系统可以获得详细的系统硬件资源信息，包括内存拓扑、CPU 与内存节点之间的关系。ACPI 表格存储在固定位置，并可以在内核启动时解析。

## 3. 内存检测 (Memory Probing)

- 操作系统可以尝试通过探测的方式逐步写入和读取不同的物理内存地址，探测哪些地址是可访问的。这种方法虽然可以用来检测系统的可用物理内存范围，但通常不推荐使用，因为可能会与系统保留内存或设备内存冲突。
- 这种方法通常只在早期操作系统（如 DOS）中使用，现在更多依赖于硬件提供的标准接口来获取内存布局。

## 4. 与 Bootloader 协作

在现代系统中，Bootloader（引导加载程序）通常负责在加载操作系统内核之前收集物理内存信息。例如，在 Linux 内核中，`GRUB`（GRand Unified Bootloader）会在启动时获取物理内存信息，并将其传递给内核。内核通过 Bootloader 提供的内存布局表来初始化内存管理模块。

## 5. 读取设备树 (Device Tree)

在某些嵌入式系统或架构（如 ARM）中，操作系统可以通过读取设备树（Device Tree）来获取系统内存布局。设备树是一种描述硬件的结构化数据，包含了从内存、CPU 到外设的信息，通常在启动时由 Bootloader 或固件传递给操作系统。

## 6. 硬件探测工具

某些现代处理器提供了硬件支持来获取物理内存信息，例如使用特定的 CPU 寄存器或 MSR（Model-Specific Registers）。这些寄存器可以帮助操作系统直接从硬件中读取系统资源信息，包括物理内存范围。

## 7. 通过虚拟化接口

在虚拟化环境中，操作系统可以通过与虚拟化管理程序（如 KVM、Xen 或 VMware）的接口，查询虚拟化管理程序提供的虚拟内存映射。在这些情况下，虚拟机管理程序负责管理实际物理内存，操作系统在启动时可以通过指定接口获取虚拟机可用的物理内存信息。

# 知识点总结

## SV39采用三层页表结构

SV39是一种在RISC-V架构中实现的虚拟地址翻译机制，它使用了39位的虚拟地址和物理地址空间，适用于64位处理器。该方式通过三级页表结构，将虚拟地址映射到物理地址，实现对内存的有效管理。

**页表结构：**SV39采用三层页表结构，包括：

- **一级页表：**负责管理较大范围的虚拟地址空间，通常为512个条目。
- **二级页表：**进一步细分地址空间，提供更高的地址解析能力。
- **三级页表：**负责管理最小的页面，确保有效利用内存。
- **每个页表项 (PTE) 包含以下信息：**
  - **有效位 (V)：**指示当前页表项是否有效。
  - **可读位 (R)：**指示当前页表项指向的内存区域是否可读。
  - **可写位 (W)：**指示当前页表项指向的内存区域是否可写。
  - **可执行位 (X)：**指示当前页表项指向的内存区域是否可执行。
  - **用户态访问位 (U)：**指示当前页表项是否允许用户态访问。
  - **系统全局位 (G)：**指示当前页表项是否为系统全局页表项。
  - **访问时间戳 (A)：**记录页表项的最后访问时间。
  - **脏页标志 (D)：**指示页表项是否已被修改。

**页大小：**SV39支持的页面大小通常为4KB，提供灵活的内存分配。也可以通过其他配置支持更大的页面，如2MB和1GB的页。

**虚拟地址结构：**

- **高位地址 (63-39位)：**用于表示虚拟页号 (VPN)，共分为三级，每级有512个可用页号。
- **低位地址 (12位)：**用于表示页内偏移量，即访问页面时的具体偏移量。

**寻址过程：**

1. **设置MMU：**通过修改 `satp` 寄存器，启用分页模式，并设置根页表的物理地址。
2. **根页表：**根据 `satp` 寄存器的设置，初始化根页表。
3. **二级页表：**根页表指向二级页表，二级页表根据虚拟地址的中间部分进行索引。
4. **三级页表：**二级页表指向三级页表，三级页表根据虚拟地址的低位部分进行索引。
5. **物理地址：**通过组合页内偏移量和三级页表的物理地址，计算最终的物理内存地址。

SV39的设计为RISC-V架构提供了灵活、高效的虚拟内存管理方案，使得它能够满足现代操作系统和应用的需求。

## 不同内存管理方式的对比

分段 (Segmentation) 和分页 (Paging) 是两种常用的内存管理机制，它们各自有不同的设计理念和实现方式。

## 1. 基本概念

- **分页 (Paging) :**
  - 将虚拟地址空间划分为固定大小的页（通常为4KB），并将物理内存划分为相同大小的页框（Page Frame）。
  - 每个进程有一个页表，用于维护虚拟页到物理页框的映射。
  - 通过页表，操作系统可以将非连续的虚拟地址空间映射到物理内存中。
- **分段 (Segmentation) :**
  - 将虚拟地址空间划分为不同大小的段（Segment），每个段代表一个逻辑单元（如代码段、数据段、堆栈段等）。
  - 每个进程有一个段表，包含每个段的起始地址和长度。
  - 段的大小可以不固定，反映了程序的逻辑结构。

## 2. 地址结构

- **分页:**
  - 虚拟地址分为两部分：页号和页内偏移。
  - 例如，对于4KB的页，虚拟地址的高位用于表示页号，低位用于表示页内的偏移（12位偏移）。
- **分段:**
  - 虚拟地址分为段号和段内偏移。
  - 段号用于索引段表，段内偏移则表示在该段内的位置。

## 3. 内存管理方式

- **分页:**
  - 分页系统消除了外部碎片（即，进程之间的空闲内存块），但可能会产生内部碎片（即，页内未使用的空间）。
  - 页表开销：每个进程需要维护页表，增加了内存占用。
- **分段:**
  - 分段系统通常更符合程序的逻辑结构，支持不同类型的内存区域。
  - 分段的灵活性使得可以在段中任意分配大小，但可能会产生外部碎片（即，随着段的创建和释放，内存中可能会出现无法利用的空闲区域）。

## 4. 权限与保护

- **分页:**
  - 页表中的每个页表项可以包含访问权限（可读、可写、可执行等），提供对内存的细粒度保护。
- **分段:**
  - 每个段也可以有自己的权限设置，允许更灵活的保护策略，因为段通常代表一个逻辑单位。

## 5. 优缺点

### 分页优点：

- 消除外部碎片。
- 进程可以使用比实际物理内存更大的地址空间（支持虚拟内存）。
- 管理相对简单，适用于多任务环境。

### 分页缺点：

- 可能导致内部碎片（尤其是当页大小不适合数据时）。
- 页表的管理和查找开销相对较高。

### 分段优点：

- 更符合程序逻辑，支持不同大小的逻辑单元。
- 允许灵活的内存分配，可以减少外部碎片。

### 分段缺点：

- 可能会产生外部碎片，特别是在长期运行中。
- 段表的管理复杂度较高，尤其是在段频繁变化的情况下。

## 6. 适用场景

### • 分页：

- 更适合需要频繁上下文切换的多任务系统，尤其是在需要支持虚拟内存的情况下。

### • 分段：

- 更适合对程序逻辑结构有较强需求的应用，如大型软件系统、操作系统内核等。