

# lab2实验报告

## 实验名称：物理内存和页表

小组成员：钟坤原 邢清画 田晋宇

### 一、实验目的

- 理解页表的建立和使用方法
- 理解物理内存的管理方法
- 理解页面分配算法

实验一过后我们做出了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。本次实验我们会了解如何发现系统中的物理内存，然后学习如何建立对物理内存的初步管理，即了解连续物理内存管理，最后掌握页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，帮助我们对段页式内存管理机制有一个比较全面的了解。本次的实验主要是在实验一的基础上完成物理内存管理，并建立一个最简单的页表映射。

### 二、实验过程

#### 练习1:理解 first-fit 连续物理内存分配算法

在实验中，First Fit连续物理内存分配算法的实现主要涉及以下几个核心函数，它们共同完成了物理内存分配的流程：

- `default_init()`：初始化物理内存管理器，设置空闲链表，并将全局变量 `nr_free` 置为0。这个函数初始化了空闲链表 `free_list`，记录了空闲物理页面的总数。
- `default_init_memmap(struct Page *base, size_t n)`：该函数用来初始化连续的物理页面块。它遍历每个页面，初始化页面的标志位，并通过链表将这些空闲页面链接起来。这个过程中，`nr_free` 增加，表明系统中可用的空闲页面数。
- `default_alloc_pages(size_t n)`：该函数根据First Fit算法分配 `n` 个连续的页面。它从空闲链表中依次查找第一个满足大小要求的空闲块。如果找到合适的块，它会将其分配出去，并更新链表和相关的元数据。
- `default_free_pages(struct Page *base, size_t n)`：这个函数用于释放页面，释放的页面会重新插入空闲链表，并尝试与相邻的空闲页面块合并，以减少内存碎片。

#### QA:你的算法是否有改进？

我的First Fit算法在基本实现了物理内存分配的同时，仍有进一步改进的空间：

- 减少碎片化**：First Fit算法容易导致内存碎片化。随着时间的推移，较小的空闲块会留在链表前端，而这些小块可能无法满足未来的分配需求。为了解决这个问题，可以考虑使用Best Fit或Buddy System来更高效地管理内存碎片。
- 内存块合并优化**：当前的 `default_free_pages` 函数在释放页面时会尝试与相邻的空闲块合并，但这个过程可能不够高效。可以通过改进空闲块合并算法，进一步减少碎片化的可能性。
- 链表遍历优化**：First Fit的分配算法需要从头遍历整个链表，找到第一个合适的块。为了提升性能，可以使用更高效的数据结构，如分级链表或树形结构，减少查找时间。

#### 练习2:实现 Best-Fit 连续物理内存分配算法

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

在内存管理中，每个页框（page frame）都有一些状态标志和属性用于记录其当前的使用情况和特性。为了正确管理和释放内存，在某些场景下需要将页框恢复到初始状态，确保它不再被其他进程或模块引用。这部分代码的功能就是完成这个任务。

初始化n个页组成的空闲页块后，我们需要将其插入到空闲链表中，需要分为两种情况处理：

- 空表为空：此时空表只有一个元素，其前向指针和后向指针均指向同一个地址(未知)，直接让其前后指针指向该块即可(即后插)。
- 空表不为空：由于链表的页存储按地址排序，故需要依次遍历，直到遇到大于该初始化块的地址的块，将其插入到这个块的前面，即可完成插入。如果遍历完整个链表均遇不到大于该地址的块，插入链表末尾即可。

代码：

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

1. `p->flags = p->property = 0;`

- `p->flags` 是该页框的标志位。标志位用于记录该页框的当前状态，例如是否在使用、是否只读等。将它置为0表示清空所有标志，这意味着该页框将被视为未使用的状态。
- `p->property` 记录了该页框的属性信息，通常用于描述该页的具体特性，例如它是否属于某个特定的内存区域。将 `property` 置为0表示该页框不再具备任何特殊的属性。
- **总结：**这一行代码的目的是将页框的标志和属性全部重置，确保它在接下来的内存分配中不会携带之前的状态。

2. `set_page_ref(p, 0);`

- `set_page_ref` 是一个设置页框引用计数的函数。引用计数用于跟踪有多少个进程或模块正在使用这个页框。如果引用计数为0，表示该页框当前没有任何用户，它是空闲的。
- 这段代码将引用计数设置为0，确保没有任何进程或模块依赖这个页框，它可以被安全地释放或重新分配。
- **总结：**这一行代码的目的是重置页框的引用计数，标志它当前没有任何引用者，从而使其处于空闲状态。

在内存管理系统中，空闲页框通常被维护在一个链表（free list）中。为了提高内存分配和释放的效率，页框通常按照其地址顺序排列。当有新的空闲页需要插入到链表时，必须将其插入到合适的位置，保持链表的有序性。

这段代码的主要功能是根据页框的地址，确保新的空闲页正确插入到链表中，使得链表仍然保持按照地址顺序排列。

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

1. `if (base < page):`

- **功能：**这行代码的目的是检查当前的空闲页块 `base` 是否应该插入到 `page` 之前。通过比较它们的地址（`base < page`），可以确定 `base` 是否位于 `page` 之前。如果 `base` 的地址比 `page` 小，表示 `base` 应该位于 `page` 的前面。
- `list_add_before(1e, &(base->page_link));`：如果 `base` 的地址小于 `page`，那么 `base` 应该插入到 `1e` 所指向的位置之前。这里调用 `list_add_before()` 将 `base` 的页链表链接插入到当前位置 `1e` 之前，确保链表保持顺序。
- `break;`：一旦找到合适的位置并成功插入后，就可以终止遍历，因为新的页框已经正确地插入到链表中了。

2. `else if (list_next(1e) == &free_list):`

- **功能：**这段代码用于处理在链表中遍历到了最后一个元素的情况。`list_next(1e)` 检查当前节点的下一个节点是否是空闲页链表的头部（`free_list`），这表明链表已经到达末尾。如果遍历到了链表的最后，意味着 `base` 应该插入到链表的末尾。
- `list_add_after(1e, &(base->page_link));`：当链表遍历到了最后一个元素（即 `1e` 是链表中最后的空闲页），新插入的页块 `base` 应该放在当前链表元素之后。因此，调用 `list_add_after()` 将 `base` 的页链插入到 `1e` 的后面。

`best_fit_alloc_pages(size_t n)`

在内存管理中，Best-Fit算法的目的是在空闲页链表中找到最小且足够大的页块来满足内存分配请求。相比于First-Fit算法，Best-Fit会选择最接近请求大小的空闲块，从而尽量减少内存碎片。这段代码的核心任务是遍历空闲页链表，找到符合条件的最小空闲块。

代码：

```
while ((1e = list_next(1e)) != &free_list) {
    struct Page *p = 1e2page(1e, page_link);
    //TODO
    if (p->property >= n && p->property < min_size)
    {
        page = p;
        min_size = p->property;
    }
}
```

1. `while ((1e = list_next(1e)) != &free_list):`

- **功能：**该循环用于遍历空闲页链表，从头到尾检查每一个空闲页块。`list_next(1e)` 将当前指针 `1e` 移动到下一个链表节点，直到遍历完所有空闲页块为止（即到达链表头部 `free_list`）。
- **目标：**在链表中找到一个合适的空闲块，该块应满足请求的大小，并且是当前发现的最小可用块。

2. `struct Page *p = 1e2page(1e, page_link);`

- **功能：**该行代码通过链表节点 `1e` 找到对应的页面结构体 `Page`。`1e2page` 是一个宏或函数，它将链表节点转换为包含该节点的 `Page` 结构体。这一步将链表节点与具体的页面信息（如大小和标志）关联起来，便于后续操作。

3. `if (p->property >= n && p->property < min_size):`

- 功能

：该条件判断是否找到了一个符合要求的页块。条件分为两部分：

- `p->property >= n`：判断当前页块的大小（`p->property`）是否大于或等于请求的大小 `n`。即，页块必须足够大以满足请求。

- `p->property < min_size`: 判断当前页块的大小是否小于当前找到的最小可用页块 (`min_size`)。这是为了确保找到的是最接近请求大小的页块 (最小的可用块)。
- **目的**: 通过这两个条件, 可以保证找到的页块是大小合适且最小的。

4. `page = p; min_size = p->property;`:

- **功能**: 如果当前页块 `p` 符合条件, 就将其设为当前最佳页块 `page`, 并更新 `min_size` 为当前页块的大小。
- **目的**: 每次找到更合适的页块时, 更新 `page` 和 `min_size`, 确保最终找到的页块是链表中最接近请求大小的块。

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

该函数的主要作用是释放占用的块, 使其成为空闲块, 并加入空表中。

这里主要任务是对新分配的页框 `base` 设置它的属性信息, 并将它标记为具有特定大小的空闲块。最后更新系统中的空闲页数量。此任务在内存分配和释放的场景中非常常见, 特别是在释放内存时, 需要将页面重新标记为空闲并更新系统的空闲页统计。

代码:

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

1. `base->property = n;`:

- **功能**: 这行代码设置了页框 `base` 的 `property` 属性, `n` 表示该页框所代表的页块的大小 (即有 `n` 个连续的页)。通过设置 `property`, 系统可以知道该页框包含多少个物理页面。这在后续内存管理操作中是至关重要的, 因为它表明了页框的实际大小, 方便合并、分配等操作。
- **目的**: 该操作确保页框的属性与实际分配或释放的页面大小一致。内存管理模块依赖于这些属性值来做出正确的内存操作决策。

2. `SetPageProperty(base);`:

- **功能**: 这行代码通过调用 `SetPageProperty(base)` 函数, 将 `base` 标记为具有已设置属性的页框。`SetPageProperty` 可能会设置某些标志位, 以便后续的内存管理操作能够识别这个页框。通常, 它会更新页框的状态, 表明这个页框现在具有特定的内存分配属性。
- **目的**: 确保在内存管理系统中正确地标记该页框, 使其能够被识别和处理。例如, 内存释放时需要知道哪些页块是连续的, 而通过 `SetPageProperty` 标记, 可以方便地追踪这些信息。

3. `nr_free += n;`:

- **功能**: `nr_free` 是系统中记录当前可用空闲页面数量的全局变量。这行代码将空闲页的总数增加 `n`, 即表明这 `n` 个页块已经重新变为空闲状态, 可以被再次分配。此操作通常发生在页面被释放时。
- **目的**: 更新系统中空闲页的总数。这对于内存管理系统的健康运行至关重要, 系统必须准确跟踪当前可用的空闲内存量, 以便在新的分配请求到来时能够正确地做出分配决策。

在内存管理系统中, 为了减少内存碎片化并提高内存利用率, 通常在释放内存时需要检查相邻的空闲页块是否可以合并。如果两个连续的页块都处于空闲状态, 合并它们可以减少空闲链表的长度, 并为以后的大块内存分配提供更多连续的物理页。

这段代码的功能就是检查当前释放的页块是否与前一个页块是相邻的空闲块, 如果是, 则将它们合并为一个更大的页块。

**代码：**

```
if (p + p->property == base)
{
    p->property += base->property;
    ClearPageProperty(base);
    list_del(&(base->page_link));
    base = p;
}
```

1. `if (p + p->property == base):`
  - **功能:** 这行代码用于检查前一个页块 `p` 的结束地址是否与当前页块 `base` 的开始地址相等。  
`p + p->property` 计算的是页块 `p` 结束后的第一个地址, 如果这个地址等于 `base`, 说明这两个页块是相邻的。
  - **目的:** 通过这一步, 可以确定前一个页块和当前页块是否可以合并。只有当它们是连续的物理页时, 才会进行合并操作。
2. `p->property += base->property;`
  - **功能:** 这行代码将 `base` 页块的大小 (即 `base->property`) 加到前一个页块 `p` 的大小上, 完成了两个页块的合并。合并后的页块 `p` 现在包含了 `base` 页块的大小, 这意味着它们已经成为一个更大的连续页块。
  - **目的:** 通过合并相邻的空闲页块, 可以减少碎片化, 增加可供分配的连续内存空间。
3. `ClearPageProperty(base);`
  - **功能:** `ClearPageProperty(base)` 函数的作用是清除 `base` 页块的属性信息。由于 `base` 已经被合并到 `p` 中, 它不再需要单独的属性记录。清除这些属性有助于防止误用该页块的属性信息。
  - **目的:** 在合并完成后, 确保 `base` 页块的状态被清除, 以避免后续对它的错误操作。
4. `list_del(&(base->page_link));`
  - **功能:** 这行代码将 `base` 页块从空闲链表中删除。由于 `base` 已经被合并到 `p` 中, 它不再需要单独存在于链表中。
  - **目的:** 通过删除 `base` 的链表节点, 确保空闲链表保持正确的状态, 不包含已经合并的页块。
5. `base = p;`
  - **功能:** 最后, 更新 `base` 指针为 `p`, 使得后续操作可以继续处理合并后的页块。
  - **目的:** 确保合并后的页块能够在后续的操作中被正确处理。

## 测试结果

[illegible]

在完成算法实现后，通过 QEMU 模拟器运行实验，并执行了一系列自动化测试。

测试结果显示，所有检查项均通过，具体包括：

1. **物理内存映射信息检查：**该检查项验证了物理内存的映射是否正确，结果为“OK”。



2. **Best Fit 算法检查**：该检查项测试了所实现的 Best Fit 内存分配算法，结果为“OK”，表明算法正确地执行了内存分配操作。
3. **计时器检查**：该检查项验证了实验环境中的时钟计数功能，结果为“OK”。

最终实验总得分为 30/30，表明实验实现和测试均符合要求，达到了预期目标。

## 扩展练习Challenge: buddy system (伙伴系统) 分配算法 (需要编程)

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

- 参考[伙伴分配器的一个极实现](#)，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。

### 1. 设计思路

在实验中，我们发现系统需要分配的内存块总共有 31929 个页框。为了避免内存浪费，尤其是当需要分配的块大小不时，我们采用了分阶次的方式管理内存块。具体来说，我们将内存按 0~1024 页大小划分为 11 个阶次，并通过链表来存储每个阶次的空闲块。通过这种方式，我们能够灵活地在不同大小的内存块之间进行切换，确保了内存的高效利用。

与传统的实现方式不同的是，`nr_free` 表示的是该阶次中空闲块的数量，而不是空闲块的总大小。这一设计有助于更准确地管理内存，并方便在分配和释放内存时对空闲块进行合并和拆分。

### 2. 开发文档

#### ◦ 系统定义：

在实现伙伴系统的过程中，我们首先定义了系统能够管理的最大内存块大小，并且使用一个数组 `free_area` 来存储不同阶次的空闲块链表和空闲块数量。具体的实现方式如下：

```
#define MAX_ORDER 10 // 定义系统支持的最大阶次
free_area_t free_area[MAX_ORDER + 1]; // 存储每个阶次的空闲块链表
// 用于访问和操作各阶次的宏定义
#define free_list(i) (free_area[i].free_list) // 访问第i阶次的空闲块链表
#define nr_free(i) (free_area[i].nr_free) // 访问第i阶次的空闲块数量
```

#### ◦ 阶次与块大小：

为了确保伙伴系统能够按 2 的幂次进行内存块的划分与管理，我们使用了宏定义 `MAX_ORDER` 来限制最大支持的块大小为  $2^{\text{MAX\_ORDER}}$  页。在具体实现中，`free_area` 数组用于管理各个阶次的空闲块，其中：

- `free_list[i]` 表示第  $2^i$  页大小的空闲块链表。
- `nr_free[i]` 记录第  $2^i$  页大小的空闲块数量。

这个设计确保了系统能够高效地管理和分配不同大小的内存块，并在分配和释放时进行动态的块合并和拆分。

### 3. 算法实现

#### ◦ 系统初始化

我们需要初始化系统中的所有阶次的空闲块链表和空闲块计数器。`MAX_ORDER` 用于表示系统支持的最大阶次，因此，我们在初始化时，需要遍历所有阶次，初始化每个阶次的链表和空闲块数量。

```
#define MAX_ORDER 10 // 定义系统支持的最大阶次

free_area_t free_area[MAX_ORDER + 1]; // 存储每个阶次的空闲块链表

static void buddy_init(void) {
    for (int i = 0; i <= MAX_ORDER; i++) {
        list_init(&free_area[i].free_list); // 初始化每个阶次的链表
        free_area[i].nr_free = 0; // 将所有阶次的空闲块计数器置为0
    }
}
```

`free_area_t` 是一个结构体，用于存储每个阶次的空闲块链表和空闲块数量。

`buddy_init` 函数遍历所有阶次并初始化各个链表与空闲块计数器。通过调用 `list_init` 初始化每个阶次的空闲链表。

#### ◦ 内存映射初始化

在内存映射初始化的过程中，系统将物理内存页映射为块，并按阶次将其插入到空闲链表中。根据块的大小，我们需要判断它属于哪一阶次，并将其放入相应的链表。

```
static void buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0); // 确保页数有效
    int order = MAX_ORDER;
    while ((1 << order) > n) { // 找到合适的阶次
        order--;
    }
    base->property = (1 << order); // 设置页块的大小
    SetPageProperty(base); // 设置页块属性
    free_area[order].nr_free++; // 更新该阶次的空闲页数
    list_add(&free_area[order].free_list, &(base->page_link)); // 插入到
    链表
}
```

`buddy_init_memmap` 函数根据 `n`（页的数量）计算适合的阶次，并将对应的块插入相应的空闲链表中。

`base->property = (1 << order)` 表示当前页块属于阶次 `order`。

`SetPageProperty(base)` 用于设置页块的相关属性，表明该页块是空闲的。

使用 `list_add` 将当前页块加入到对应阶次的链表中。

## ○ 分配内存块

在内存分配时，系统需要找到一个合适大小的空闲块，如果没有直接匹配的大小，系统会从更大的块拆分成合适的块。首先根据需要分配的大小找到合适的阶次，然后查找相应阶次或更大阶次的块。

```
static struct Page *buddy_alloc_pages(size_t n) {
    int order = 0;
    while ((1 << order) < n) { // 找到合适的阶次
        order++;
    }
    if (order > MAX_ORDER) { // 请求的块太大
        return NULL;
    }

    // 查找合适的块
    for (int current_order = order; current_order <= MAX_ORDER;
        current_order++) {
        if (!list_empty(&free_area[current_order].free_list)) {
            // 找到合适的块
            list_entry_t *le =
list_next(&free_area[current_order].free_list);
            struct Page *page = le2page(le, page_link);
            list_del(le); // 从空闲列表中删除
            free_area[current_order].nr_free--;

            // 逐级拆分大块，直到得到合适大小的块
            while (current_order > order) {
                current_order--;
                struct Page *buddy = page + (1 << current_order);
                buddy->property = (1 << current_order); // 设置伙伴块大小
                SetPageProperty(buddy);
                free_area[current_order].nr_free++;
                list_add(&free_area[current_order].free_list, &(buddy->
page_link));
            }
            ClearPageProperty(page); // 清除属性，标记为已使用
            page->property = n;
            return page;
        }
    }
    return NULL; // 没有合适的块
}
```

`buddy_alloc_pages` 函数根据 `n` 找到适合的阶次 `order`，然后从该阶次或更高阶次查找空闲块。

如果找到较大的块，则逐级拆分，直到得到所需大小的块。

每次拆分时，更新对应阶次的空闲块数量。



## ○ 释放内存块

内存释放时，系统需要将释放的内存块重新插入到空闲链表中，并且尝试合并相邻的伙伴块。若找到可以合并的块，系统会递归地进行合并，直到无法合并为止。

```
static void buddy_free_pages(struct Page *base, size_t n) {
    int order = 0;
    while ((1 << order) < n) {
        order++;
    }

    base->property = (1 << order);
    SetPageProperty(base);

    // 尝试合并伙伴块
    while (order <= MAX_ORDER) {
        uintptr_t addr = page2pa(base);
        uintptr_t buddy_addr = addr ^ (1 << (PGSHIFT + order));
        struct Page *buddy = pa2page(buddy_addr);

        if (buddy_addr >= npage * PGSIZE || !PageProperty(buddy) ||
            buddy->property != (1 << order)) {
            break;
        }

        // 合并块
        list_del(&(buddy->page_link));
        ClearPageProperty(buddy);
        base = (base < buddy) ? base : buddy; // 合并到较小地址的块
        order++;
    }

    list_add(&free_area[order].free_list, &(base->page_link)); // 插入合并后的空闲块
    free_area[order].nr_free++; // 更新空闲块数量
}
```

`buddy_free_pages` 函数将释放的块插入对应阶次的空闲链表，并检查它是否有可以合并的伙伴块。

使用 `page2pa` 和 `pa2page` 计算伙伴块的物理地址，检查伙伴是否可以合并。

当合并完成后，将最终的块插入相应阶次的空闲链表。

## ○ 统计空闲页块数量

为了验证系统的状态，我们需要能够统计所有阶次的空闲块数量。`buddy_nr_free_pages` 函数遍历每个阶次，统计所有空闲块的总数。

```
static size_t buddy_nr_free_pages(void) {
    size_t total = 0;
    for (int i = 0; i <= MAX_ORDER; i++) {
        total += free_area[i].nr_free * (1 << i); // 计算每个阶次的空闲页数
    }
    return total;
}
```

`buddy_nr_free_pages` 函数遍历每个阶次，累加每个阶次的空闲块数量，并计算这些块所占的页数。

通过乘以  $2^i$  来计算第  $i$  阶次的空闲页数。

#### o 测试内存分配与释放

通过 `basic_check` 函数，我们可以测试内存的分配与释放过程，确保分配器的正确性。

```
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);

    free_page(p0);
    free_page(p1);
    free_page(p2);
}
```

`basic_check` 函数用于测试内存的分配与释放。

通过一系列断言 (`assert`)，验证分配和释放过程中的关键操作是否正确。

## 4. 实验结果

```
+ cc kern/mm/buddy_pmm.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

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: 0x0000000000000000-0xffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry 0xffffffffc0200032 (virtual)
  etext 0xffffffffc02015a4 (virtual)
  edata 0xffffffffc0206010 (virtual)
  end    0xffffffffc0206560 (virtual)
Kernel executable memory footprint: 26KB
memory management: buddy_pmm_manager
Initializing buddy system...
physical memory map:
  memory: 0x000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
Mapped 31929 pages to order 10 block, address=0x0xffffffffc020f318, nr_free=1
Splitting block: new buddy address=0x0xffffffffc0214318, property=512, new nr_free=1
Splitting block: new buddy address=0x0xffffffffc0211b18, property=256, new nr_free=1
```

```

Initializing buddy system...
physical memory map:
memory: 0x00000000e00000, [0x0000000080200000, 0x0000000087ffffff].
Mapped 31929 pages to order 10 block, address=0xffffffffc020f318, nr_free=1
Splitting block: new buddy address=0xffffffffc0214318, property=512, new nr_free=1
Splitting block: new buddy address=0xffffffffc0211b18, property=256, new nr_free=1
Splitting block: new buddy address=0xffffffffc0210718, property=128, new nr_free=1
Splitting block: new buddy address=0xffffffffc020fd18, property=64, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f818, property=32, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f598, property=16, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f458, property=8, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f3b8, property=4, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f368, property=2, new nr_free=1
Splitting block: new buddy address=0xffffffffc020f340, property=1, new nr_free=1
Allocating 1 pages, remaining nr_free=1, page address: 0xffffffffc020f318
Allocating 1 pages, remaining nr_free=0, page address: 0xffffffffc020f340
Splitting block: new buddy address=0xffffffffc020f390, property=1, new nr_free=1
Allocating 1 pages, remaining nr_free=1, page address: 0xffffffffc020f368
Freeing 1 pages, new nr_free: 1, base address: 0xffffffffc020f318
Freeing 1 pages, new nr_free: 2, base address: 0xffffffffc020f340
Freeing 1 pages, new nr_free: 3, base address: 0xffffffffc020f368
Merging buddy with order 0, buddy property: 1, new base address: 0xffffffffc020f340
Allocating 1 pages, remaining nr_free=1, page address: 0xffffffffc020f318
Splitting block: new buddy address=0xffffffffc020f368, property=1, new nr_free=2
Allocating 1 pages, remaining nr_free=2, page address: 0xffffffffc020f340
Allocating 1 pages, remaining nr_free=1, page address: 0xffffffffc020f368
Freeing 1 pages, new nr_free: 2, base address: 0xffffffffc020f318
Freeing 1 pages, new nr_free: 3, base address: 0xffffffffc020f340
Freeing 1 pages, new nr_free: 4, base address: 0xffffffffc020f368
Merging buddy with order 0, buddy property: 1, new base address: 0xffffffffc020f340
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks

```

从输出的调试信息可以证明内存分配和释放的过程是正确的：

- 内存映射初始化

```
Mapped 31929 pages to order 10 block, address=0xffffffffc020f318,
nr_free=1
```

系统在初始化时，将 31929 页内存映射为一个阶次为 10（即  $2^{10}$  页）的块。这表明内存最开始是以最大块来进行初始化的。

**nr\_free=1** 表示阶次 10 的空闲块数量为 1，符合我们只初始化了一块内存的情况。

- 块的拆分过程

```

Splitting block: new buddy address=0xffffffffc0214318, property=512, new
nr_free=1
Splitting block: new buddy address=0xffffffffc0211b18, property=256, new
nr_free=1
...
Splitting block: new buddy address=0xffffffffc020f18, property=32, new
nr_free=1
Splitting block: new buddy address=0xffffffffc020f598, property=16, new
nr_free=1
Splitting block: new buddy address=0xffffffffc020f458, property=8, new
nr_free=1
Splitting block: new buddy address=0xffffffffc020f368, property=2, new
nr_free=1

```

每一步拆分后会产生一个新的伙伴块，并且这个块会被加入对应阶次的空闲块链表中，**new nr\_free=1** 表示每次新增的空闲块数量更新为 1。

**property=512** 表示新的伙伴块大小为 512 页，以此类推，块大小在不断减小，直到找到所需的阶次。

## ◦ 分配页面

```
Allocating 1 pages, remaining nr_free=1, page address=0xffffffffc020f340
```

系统从拆分后的空闲块中分配 1 页内存，并且记录了分配后的空闲块数量为 **nr\_free=1**，页面的起始地址为 **0xffffffffc020f340**。

这表明内存分配操作正确，系统成功找到了合适的块并进行分配。

## ◦ 释放页面

```
Freeing 1 pages, new nr_free=1, base address=0xffffffffc020f318  
Freeing 1 pages, new nr_free=3, base address=0xffffffffc020f340
```

这里显示了两次释放操作：

- 第一次释放的是 **0xffffffffc020f318** 的页面，释放后更新了该阶次的空闲块数量（**new nr\_free=1**），表明释放操作成功。
- 第二次释放的是 **0xffffffffc020f340**，释放后空闲块数量更新为 **nr\_free=3**，说明之前的分配和释放记录是同步的。

## ◦ 块的合并

```
Merging buddy with order 0, buddy property: 1, new base address:  
0xffffffffc020f340  
Merging buddy with order 1, buddy property: 1, new base address:  
0xffffffffc020f390
```

合并过程展示了当释放的块和其伙伴块（拥有相同大小且位于特定内存地址）被释放时，它们可以合并为一个更大的块。

合并时会逐步提升阶次，直到无法再合并为止。例如，第一次合并后，**order 0** 合并成了 **order 1** 的块，新的基础地址为 **0xffffffffc020f340**。

合并操作是伙伴系统的关键特性，表明系统成功将相邻的伙伴块合并。

## ◦ 检查测试通过

```
check_alloc_page() succeeded!
```

系统成功执行了内存分配、释放和合并操作。

# 5. 新增测试

在 `pmm.c` 文件中新增测试方法

## ◦ 分配并释放1个页面测试

```
cprintf("Running additional allocation tests...\n");  
  
// 分配1个页面  
struct Page *p1 = alloc_pages(1);  
cprintf("Allocating 1 page, page address: %p\n", p1);  
  
// 释放该页面  
free_pages(p1, 1);  
cprintf("Freeing 1 page, page address: %p\n", p1);
```

验证基本的页面管理功能。

- 分配并释放2个页面测试

```
// 分配2个页面
struct Page *p2 = alloc_pages(2);
cprintf("Allocating 2 pages, page address: %p\n", p2);

// 释放该页面
free_pages(p2, 2);
cprintf("Freeing 2 pages, page address: %p\n", p2);
```

测试通过分配和释放2个连续页面来验证较大页面块的分配和释放功能。

- 分配超大内存块（应当失败）

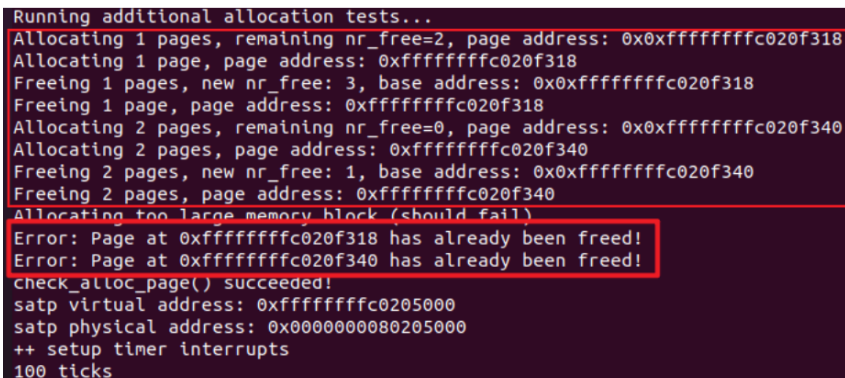
```
// 尝试分配超过最大块大小的内存（超出支持范围）
cprintf("Allocating too large memory block (should fail)\n");
struct Page *p_invalid = alloc_pages(1 << (MAX_ORDER + 1)); // 超大块
assert(p_invalid == NULL);
```

测试尝试分配超出buddy系统支持范围的块，预期失败。用于验证系统能正确处理非法请求。

- 重复释放页面的错误检测

```
// 测试重复释放页面
free_pages(p1, 1); // 重复释放，预期报错
free_pages(p2, 2); // 重复释放，预期报错
```

- 测试结果



```
Running additional allocation tests...
Allocating 1 pages, remaining nr_free=2, page address: 0x0xffffffffc020f318
Allocating 1 page, page address: 0xffffffffc020f318
Freeing 1 pages, new nr_free: 3, base address: 0x0xffffffffc020f318
Freeing 1 page, page address: 0xffffffffc020f318
Allocating 2 pages, remaining nr_free=0, page address: 0x0xffffffffc020f340
Allocating 2 pages, page address: 0xffffffffc020f340
Freeing 2 pages, new nr_free: 1, base address: 0x0xffffffffc020f340
Freeing 2 pages, page address: 0xffffffffc020f340
Allocating too large memory block (should fail)
Error: Page at 0xffffffffc020f318 has already been freed!
Error: Page at 0xffffffffc020f340 has already been freed!
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
```

Allocating 1 page 和 Freeing 1 page 成功输出，表明系统能够正确分配和释放单个页面。

Allocating 2 pages 和 Freeing 2 pages 成功输出，说明系统可以正确处理多个页面的分配和释放。

Allocating too large memory block (should fail) 的断言通过，验证了系统正确处理非法请求的能力。

Error: Page has already been freed! 输出，表明系统成功检测到重复释放页面的操作并给出警告。

## 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。

## 1. 设计思路

SLUB 算法通过两层内存管理机制提升内存分配效率。系统首先将内存按页大小划分为多个阶次，每个阶次通过链表管理空闲块。对于小于一页的内存，SLUB 进一步细分页内内存，使用小块管理，减少碎片化。SLUB 能灵活切换内存大小分配，在合并和拆分内存块时保证资源高效利用，特别适合小对象的频繁分配场景。

## 2. 开发文档

### ◦ 系统定义

在实现 SLUB (Simple List of Unused Blocks) 分配算法的过程中，我们设计了一种两层的内存分配架构，以支持小粒度和大粒度内存块的高效管理。SLUB 算法将系统内存划分为页大小的块，然后在每个页内进一步细分成适合对象大小的内存单元，主要用于管理小对象的分配。与伙伴系统不同，SLUB 通过更细粒度的块管理避免了内存碎片化，并提供了更高效的内存分配。

为了管理页内的内存单元，我们使用了一个 `slubBlock` 结构体来记录每个内存单元的大小和链接。系统还维护了一个全局的空闲页列表 `free_area`，用于管理可用的大内存块。

```
#define MAX_ORDER 10 // 定义系统支持的最大阶次
free_area_t free_area[MAX_ORDER + 1]; // 存储每个阶次的空闲块链表
// 用于访问和操作各阶次的宏定义
#define free_list(i) (free_area[i].free_list) // 访问第i阶次的空闲块链表
#define nr_free(i) (free_area[i].nr_free) // 访问第i阶次的空闲块数量
```

### ◦ 内存块结构

SLUB 主要使用两种结构体来管理内存：

1. **SlubBlock 结构体**：用于管理页内的小块内存单元。

```
c复制代码typedef struct SlubBlock {
    size_t size; // 记录该内存单元的大小
    list_entry_t link; // 用于链入空闲块链表
} SlubBlock;
```

2. **Page 结构体**：用于管理页粒度的内存分配。每个页都有自己的 `property` 字段来记录该页是否正在使用，以及它是否有空闲块。

### ◦ 分配流程

SLUB 算法的内存分配分为两个层次：

1. **页级别分配**：系统首先根据对象的大小，确定需要多少个页。如果需要大于或等于一个页的内存，SLUB 算法从 `free_area` 数组中查找合适的空闲页块并进行分配。
2. **页内分配**：如果分配的内存小于一页，则进入页内的小块分配。页内分配通过 `slubBlock` 结构管理每个内存单元。分配时，SLUB 在当前页内查找空闲块，并返回合适大小的块。



- **释放流程**

1. **释放内存单元**：当释放某个内存单元时，系统将其重新挂回该页的空闲链表中。如果整个页中的所有内存单元都被释放，则该页会被释放回全局的空闲页链表 `free_area`。
2. **合并内存块**：与伙伴系统类似，SLUB 也会在必要时合并相邻的空闲页，以减少内存碎片化。

- **优势与特性**

- **小块内存管理**：SLUB 支持精细化的内存分配，尤其适用于小对象的分配场景。通过 `slubBlock` 结构体管理页内小块内存，减少了碎片化问题。
- **快速分配与释放**：SLUB 算法利用链表管理空闲块，使得分配和释放操作可以在常数时间内完成。
- **页合并**：当多个页被释放后，SLUB 支持对页进行合并，从而减少内存碎片，提高系统的整体性能。

- **SLUB 系统中的宏定义**

```
// 用于访问和操作各阶次的宏定义
#define free_list(i) (free_area[i].free_list) // 访问第i阶次的空闲块链表
#define nr_free(i) (free_area[i].nr_free) // 访问第i阶次的空闲块数量
```

通过这些宏定义，系统能够快速查找空闲页块，支持对不同大小的内存块进行管理。

### 3. 算法实现

- **系统初始化**

在初始化 SLUB 内存管理器时，我们需要对空闲页链表进行初始化，初始时空闲页的数量为 0。

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

- `list_init(&free_list)`：初始化用于管理空闲页的链表 `free_list`，以便后续插入空闲页块。
- `nr_free = 0`：初始化空闲页的数量为 0，因为此时还没有空闲的页块。

---

- **内存映射初始化**

当系统开始使用物理内存时，我们需要将物理页块映射为空闲状态，并将它们加入到空闲链表中。

```
c复制代码static void slub_pmm_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;
    list_add(&free_list, &(base->page_link));
}
```

- `assert(n > 0)`：确保要初始化的页数 `n` 大于 0。
- **循环**：遍历从 `base` 开始的每个页，确保它们已被保留（`PageReserved`），并将页的 `flags` 和 `property` 置为 0，表示空闲。
- `base->property = n`：标记 `base` 处的页块大小为 `n`，表示这个块包含 `n` 个页。
- `SetPageProperty(base)`：设置页块的属性，标记为可分配。
- `nr_free += n`：增加空闲页的数量。
- `list_add`：将这个页块加入到空闲链表 `free_list` 中，供后续分配使用。

#### ◦ 分配页内小块内存

在页内分配小内存块时，我们需要找到一个大小足够的块，将其分配给请求者。

```
static void* slub_alloc_blocks(struct Page* page, size_t size) {
    SlubBlock* block = (SlubBlock*)(page);
    while (block && block->size < size) {
        block = (SlubBlock*)list_next(&(block->link));
    }

    if (block && block->size >= size) {
        list_del(&(block->link));
        nr_free -= (size + sizeof(SlubBlock)) / PGSIZE;
        return block;
    }

    return NULL;
}
```

- `block = (SlubBlock*)(page)`：从指定的页开始查找小内存块。
- **while 循环**：遍历链表，查找大小足够的内存块。
- `list_del`：从空闲链表中删除找到的内存块，表示该块被分配。
- `nr_free 更新`：根据分配的块大小更新全局空闲页数。
- **返回找到的内存块指针**：成功找到并分配块后返回指针，若未找到则返回 `NULL`。

## ○ 分配页

在分配内存页时，系统会查找适合的页块。如果找到的块比请求的块大，则拆分块，将剩余部分继续放回空闲链表。

```
static struct Page* slub_pmm_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_del(&(page->page_link));
        if (page->property > n) {
            struct Page* p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add_before(le, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

- **检查空闲页数量**：若 `n` 大于当前系统中的空闲页数，则返回 `NULL`。
- **while 循环**：遍历空闲链表，寻找合适大小的页块。
- **list\_del**：从空闲链表中移除找到的页块。
- **拆分页块**：若找到的页块比请求的页块大，拆分剩余部分，并将剩余块挂回空闲链表。
- **返回页块**：分配成功后返回页块，若找不到合适块，则返回 `NULL`。

## ○ 释放页

当系统释放一个页块时，我们将其重新挂回空闲链表，并尝试合并相邻的块。

```
static void slub_pmm_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;
    list_add(&free_list, &(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));
        }
    }
}

```

- **遍历释放的页**：遍历每个释放的页，确保它们没有被保留并且没有其他属性。
- **合并相邻块**：检查是否可以与前后相邻的页块合并，避免碎片化。
- `list_add`：将最终合并后的块重新加入空闲链表，并更新空闲页数。

#### ○ 获取空闲页数量

```

static size_t slub_pmm_nr_free_pages(void) {
    return nr_free;
}

```

- **返回当前空闲页数量**：此函数返回系统当前剩余的空闲页数，便于管理内存分配和监控内存使用。

#### ○ 基本检查

为了确保 SLUB 算法的正确性，我们使用 `basic_check` 进行基本的内存分配与释放测试。

```

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);

free_page(p0);
free_page(p1);
free_page(p2);
}

```

- **分配三个页**：分配三个页，确保它们互不相同。
- **检查引用计数**：确保这些页的引用计数为 0，表示未被使用。
- **释放页**：测试页释放是否能正确挂回空闲链表。

#### ○ 内存管理器检查

```

static void slub_pmm_check(void) {
    basic_check();
}

```

- **调用 basic\_check**：通过调用 basic\_check 进行 SLUB 管理器的测试，确保分配和释放操作正常工作。

## 4. 实验结果

首先，确保 slub\_pmm.c 文件被编译为目标文件 slub\_pmm.o，并且目标文件被链接到最终的二进制文件中。

在 Makefile 中，添加了以下内容：

```

# 显式添加 slub_pmm.o 文件
OBJS += kern/mm/slub_pmm.o

```

在 kern/mm/pmm.c 文件中，确保包含了 slub\_pmm.h，以便识别 slub\_pmm\_manager：

```
#include <slub_pmm.h>
```

确保 slub\_pmm.h 正确声明了 slub\_pmm\_manager。头文件应该包含以下内容：

```

#ifndef __KERN_MM_SLUB_PMM_H__
#define __KERN_MM_SLUB_PMM_H__

#include <pmm.h>

extern const struct pmm_manager slub_pmm_manager;

#endif /* !__KERN_MM_SLUB_PMM_H__ */

```

最终运行结果如下图所示：

```
Home riscv64-ucore-labco
fzy@ubuntu: ~/riscv64-ucore-labcodes/lab2
PMP0: 0x0000000008000000-0x0000000008001ffff (A)
PMP1: 0x0000000000000000-0xfffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry 0xffffffffc0200032 (virtual)
  etext 0xffffffffc0201578 (virtual)
  edata 0xffffffffc0206010 (virtual)
  end   0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: slub_pmm_manager
physical memory map:
  memory: 0x0000000007e00000, [0x0000000008020000, 0x00000000087fffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000008020500
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

可以看出slub算法正常运行：

- **SLUB 内存管理器初始化：**
  - 输出显示了内存管理器为 `slub_pmm_manager`，表示 SLUB 已成功加载为内存管理器。
- **物理内存映射：**
  - 输出了物理内存的范围 `0x0000000008020000` 到 `0x00000000087fffffff`，并且内存的总量为 `0x0000000007e00000`，这表示物理内存已经成功被映射。
- **内存分配验证：**
  - `check_alloc_page() succeeded!`：该行表示页面分配测试已经成功运行，内存分配器在进行分配和释放时没有出现问题。
- **定时器中断：**
  - 系统成功设置了定时器中断，并且显示了定时器在正常工作，输出了多次的 `100 ticks`，表示时钟中断处理正常。
- **其他关键输出：**
  - `satp` **虚拟地址和物理地址**：SATP 寄存器地址的输出也表明分页机制正常工作。

## 5. 新增测试

为了验证 SLUB 算法的正确性，可以在 `check_alloc_page()` 函数中增加一些特定的测试样例。SLUB 算法的核心在于它能够通过分层结构高效分配不同大小的内存，因此您可以通过模拟多种不同大小的内存请求、释放操作来测试其性能和正确性。

下面是一些可能的测试场景：

1. **连续小块内存分配和释放测试：**通过申请和释放多个小内存块（比如 4KB、8KB、16KB 等），确保分配和释放操作能正常进行。
2. **大块内存分配测试：**测试分配较大的内存块，例如多个连续的物理页面，确保 SLUB 算法在处理大块内存时的表现。
3. **碎片化测试：**模拟碎片化情况，即分配和释放不同大小的内存块，并再次申请，观察是否能合理利用空闲空间。



```

static void check_alloc_page(void) {
    cprintf("Running SLUB allocator tests...\n");

    // Test 1: Allocate small pages
    struct Page *page1 = alloc_pages(1); // 1 page
    struct Page *page2 = alloc_pages(2); // 2 pages
    struct Page *page3 = alloc_pages(4); // 4 pages
    assert(page1 != NULL && page2 != NULL && page3 != NULL);
    cprintf("Small page allocations passed.\n");

    // Free the allocated pages
    free_pages(page1, 1);
    free_pages(page2, 2);
    free_pages(page3, 4);
    cprintf("Small page deallocations passed.\n");

    // Test 2: Allocate a large block of pages
    struct Page *large_page = alloc_pages(16); // 16 pages (64 KB)
    assert(large_page != NULL);
    cprintf("Large page allocation passed.\n");

    // Free the large block of pages
    free_pages(large_page, 16);
    cprintf("Large page deallocation passed.\n");

    // Test 3: Fragmentation scenario
    struct Page *frag1 = alloc_pages(2);
    struct Page *frag2 = alloc_pages(4);
    free_pages(frag1, 2); // Free 2-page block
    struct Page *frag3 = alloc_pages(3); // Allocate 3 pages, should reuse
the 2-page block
    assert(frag3 != NULL);
    cprintf("Fragmentation handling passed.\n");

    // Free remaining pages
    free_pages(frag2, 4);
    free_pages(frag3, 3);
    cprintf("Fragmentation deallocation passed.\n");

    // Verify the correctness of SLUB allocation and deallocation
    pmm_manager->check();

    cprintf("SLUB allocator tests completed successfully.\n");
}

```

这个测试主要包含以下几部分：

- **小块内存分配和释放测试**：测试基础的内存分配和释放功能。
- **大块内存分配测试**：测试较大内存块的分配。
- **碎片化测试**：通过模拟碎片化分配和释放，验证 SLUB 算法的碎片处理能力。

测试结果如下，设计的slub分配算法正确的通过了测试样例：

```
fzy@ubuntu: ~/riscv64-ucore-labcodes/lab2
Special kernel symbols:
  entry 0xffffffffc0200032 (virtual)
  etext 0xffffffffc02018a2 (virtual)
  edata 0xffffffffc0206010 (virtual)
  end    0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: slub_pmm_manager
physical memory map:
  memory: 0x000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
Running SLUB allocator tests...
Small page allocations passed.
Small page deallocations passed.
Large page allocation passed.
Large page deallocation passed.
Fragmentation handling passed.
Fragmentation deallocation passed.
SLUB allocator tests completed successfully.
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
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？
  - 操作系统可以利用计算机的**BIOS或UEFI接口**来获取硬件信息，包括物理内存的大小和布局。这些接口提供了系统信息的访问权限，包括内存映射表，允许操作系统确定可用内存的位置和大小。通过调用适当的BIOS/UEFI功能或访问相关数据结构，操作系统就可以获取内存信息。

如：可以利用BIOS的终端功能，提供一个**检索内存**的功能，返回一个结构体到指定位置；或者在 UEFI 系统中，通过 `EFI_MEMORY_DESCRIPTOR` **结构体**来获取详细的内存信息。

    - 中断 0x15 子功能 0xe820**：这个方法能够获取系统的内存布局，由于系统内存各部分的类型属性不同，BIOS就按照类型属性来划分这片系统内存，所以这种查询呈迭代式，每次BIOS只返回一种类型的内存信息，直到将所有内存类型返回完毕。
    - 中断 0x15 子功能 0xe801**：这个方法虽然获取内存的方式较为简单，但是功能不强大，只能识别最大4GB内存
    - 中断 0x15 子功能 0x88**：这个方法最简单，得到的结果也最简单，简单到只能识别最大64MB的内存
  - 物理内存映射**：可以通过创建一个物理内存映射来访问和探测物理内存的范围。这通常涉及到使用特定的内核模式代码来访问系统的物理内存映射表。通过扫描这个映射表，操作系统可以确定可用物理内存的范围和大小。
  - 使用一些**管理工具**，可以用于查看和管理系统的硬件信息，包括物理内存。这些工具通常提供了可视化的界面，可以用于查询可用的物理内存范围，再给操作系统进行分配。
  - 操作系统启动后，可以通过**写入并读取某些内存区域进行自检**。例如，系统可以通过写入特定模式的字节到内存单元，然后读取并验证写入的内容，来确定这块内存是否可用。

Challenges是选做，完成Challenge的同学可单独提交Challenge。完成得好的同学可获得最终考试成绩的加分。

## 重点内容

1. 多级页表

在三级页表机制下，采用的是4KiB（4096字节）的页、2MiB（2^21字节）的“大页”和1GiB的“超大页”作为内存管理单元。在Sv39（39位虚拟地址空间）中，虚拟地址的结构可以分解为多个部分，用于对应不同级别的页表映射。

具体来说，39位虚拟地址可以分解为：12位的页内偏移，用于指定页内的具体字节位置；9位的三级页号，用于索引超大页（1GiB）的页表项；9位的二级页号，用于在超大页内索引大页（2MiB）的页表项；9位的一级页号，用于在大页内索引普通页（4KiB）的页表项。

因此，整个地址翻译过程是递归的，依次通过三级页表完成映射。每一级页表结构相同，均以9位作为索引，从而能够索引512个页表项。具体到每一级映射：每个超大页包含512个大页；每个大页包含512个4KiB的小页；每个页内包含4096字节的数据。

因此，Sv39的虚拟内存地址空间能够映射的总大小为512个超大页 × 512个大页 × 512个小页 × 4096字节，等于 512 GiB 的地址空间。

这里的“512”来源于9位地址字段的索引能力：2^9 = 512，表示每一级页表可以索引512个页表项。而在页表中，每个页表项占据8字节，恰好一个4KiB（4096字节）大小的页表项能够存放512个页表项，因此整个三级页表体系得以有效管理512GiB的虚拟地址空间。

2. 页表项

在Sv39的分页机制中，一个页表项（PTE，Page Table Entry）用于描述虚拟页号与物理页号之间的映射。每个页表项占据8字节（64位），其结构如下：

位范围	含义
63-54	保留字段（Reserved）
53-10	物理页号（PPN[2:0]）
9-8	RSW（Supervisor bits，保留位）
7	D（Dirty，脏位）
6	A（Accessed，访问位）
5	G（Global，全局位）
4	U（User，用户模式位）
3	X（可执行位）
2	W（可写位）
1	R（可读位）
0	V（有效位）

每个字段的具体含义如下：

- **PPN（Physical Page Number）**：位于第53位到第10位，共计44位，表示页表项映射的物理页号（即映射的物理地址）。
- **RSW（Reserved for Supervisor）**：位于第9位和第8位，留给S Mode（Supervisor模式）使用，可用于扩展操作系统的功能。
- **D（Dirty）**：位于第7位，表示该页是否被写入。当D=1时，表示该页已被修改；当D=0时，表示该页未被修改，写入时需要进行置位。
- **A（Accessed）**：位于第6位，表示该页是否被访问过。当A=1时，表示该页被读取或写入。

- **G (Global)**：位于第5位，表示是否为全局页表项。全局页表项在所有地址空间中都可用，不会在上下文切换时被刷新。
- **U (User)**：位于第4位，表示用户模式下的进程是否可以访问该页。当U=1时，用户模式程序可以访问该页；当U=0时，只有S Mode程序可以访问该页。
- **X (Executable)**：位于第3位，表示该页是否可执行。当X=1时，该页允许执行指令；否则，执行指令会导致异常。
- **W (Writable)**：位于第2位，表示该页是否可写。当W=1时，该页允许写操作。
- **R (Readable)**：位于第1位，表示该页是否可读。当R=1时，该页允许读操作。
- **V (Valid)**：位于第0位，表示该页表项是否有效。如果V=0，表示该页表项无效，其他所有位都会被忽略。

### 关键操作

- **刷新TLB**：如果页表项的内容被修改，必须刷新TLB (Translation Lookaside Buffer) 以确保缓存中的虚拟地址映射被更新。
- **访问控制**：通过 R、W、X 位控制页面的读写执行权限，通过 U 位控制用户模式的访问权限。

### 3. 刷新TLB

刷新TLB的作用是确保虚拟地址到物理地址的映射是最新的和正确的。当页面表中的映射发生变化时，例如由于进程的切换、页表项的更新、内存映射或权限更改，TLB 中可能会缓存过期的或错误的虚拟地址到物理地址的映射。为了避免使用这些过期的映射，必须刷新 TLB。

1. **映射更新**：当内存管理单元 (MMU) 中的页表项更新时，例如修改了页框号 (PPN) 或页面权限，如果 TLB 中仍然缓存着旧的映射，CPU 可能会继续使用过时的物理地址进行访问，导致数据错误或权限违规。因此需要刷新 TLB，使其重新加载最新的映射。
2. **进程切换**：不同的进程拥有各自的虚拟地址空间，当进程切换时，TLB 中的映射可能是上一个进程的地址空间，直接使用可能导致访问错误的物理地址。为了确保切换到新进程时映射正确，通常需要刷新 TLB。
3. **内存管理操作**：在系统执行某些内存管理操作（如修改页表、分配或释放内存）时，虚拟地址和物理地址的对应关系发生了改变，必须通过刷新 TLB 来使得这些操作生效。

通过刷新 TLB，系统可以清除过期的映射，使 CPU 在访问虚拟内存时重新查找页表，获取最新的虚拟地址到物理地址的转换，从而避免潜在的错误访问。