操作系统lab2实验报告

小组成员：

陈秋彤（2311815）徐盈蕊（2311344）杨欣瑞（2312246）

练习一：理解first-fit 连续物理内存分配算法(思考题)

主要思想

first-fit 连续物理内存分配算法，维护一个空闲的块列表，当需要内存时，我们就找到对应的一块**内存最大**的空闲块，分配给对应的进程。

过程阐述

从空闲内存块的链表上查找**第一个**大小大于所需内存的块，分配出去，回收时会**按照地址从小到大的顺序插入链表**，并且**合并**与之**相邻且连续**的空闲内存块。

详细分析 first-fit 连续物理内存分配算法的实现过程：

代码分析

default_init

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

该函数用于初始化物理内存管理器的基本数据结构，为后续内存分配做准备：
`list_init(&free_list);` 调用list_init函数初始化空闲链表

`free_list, nr_free = 0;`设置空闲页面数为0

上述list_init函数的定义

```
static inline void
list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}
```

default_init_memmap

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

该函数将一段连续的物理页面初始化为空闲块，并按照地址顺序插入到空闲链表中：

```
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
```

初始化一段连续的物理内存页面，`base`是起始页面的指针，`n`是要初始化的页面数量并且确保页面数量大于0；

`struct Page *p = base;`创建遍历指针，指向起始页面

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

遍历从 `base` 到 `base + n` 的所有页面，初始化每个页面：

`assert(PageReserved(p));`确认该页面这是被标记为"保留"状态，确保是可用的物理内存`p->flags = p->property = 0;`清空页面标志位flag和属性值property对于非块首页面，此值为0，`set_page_ref(p, 0);`设置页面引用计数为0，表示当前没有被引用

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

设置块首页面属性：

`base->property = n;`：**设置块首页面的property字段为整个连续内存块的大小**

这是first-fit算法的核心——只有每个空闲块的第一个页面记录整个块的大小

`SetPageProperty(base);`：设置块首页面的PG_property标志位，表示该页面是一个空闲块的起始页面，`nr_free += n;`：更新全局空闲页面计数器

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

插入空闲链表（按地址排序）

情况1：空闲链表为空则直接插入

情况2：链表不为空，需要按地址顺序插入

default_alloc_pages

```
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;
}
```

函数声明和初始检查：

`assert(n > 0);`确保请求分配的页面数量n大于0，这是基本合法性检查

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

检查系统中是否有足够的空闲页面
如果请求的页面数n大于当前空闲页面总数nr_free，直接返回NULL表示分配失败

`struct Page *page = NULL;`初始化返回的页面指针为NULL，表示尚未找到合适的空闲块

`list_entry_t *le = &free_list;`获取空闲链表的头节点指针，用于开始遍历

搜索合适的空闲块：
`while ((le = list_next(le)) != &free_list) {`开始遍历空闲链表，list_next(le)获取下一个节点，循环直到回到链表头

`struct Page *p = le2page(le, page_link);`将链表节点转换为对应的Page结构体指针，le2page是宏定义，通过计算结构体偏移量得到Page指针

```
if (p->property >= n) {
page = p;
break;
}
```

检查当前空闲块的大小是否满足请求，p->property记录了这个空闲块包含的页面数量
如果满足条件，将page指向这个块，并跳出循环。这里体现了first-fit策略：找到第一个满足条件的块就停止搜索

处理找到的空闲块
`if (page != NULL) {`检查是否成功找到了合适的空闲块

`list_entry_t* prev = list_prev(&(page->page_link));`记录当前块在链表中的前驱节点，为后续可能的插入操作做准备

`list_del(&(page->page_link));`从空闲链表中删除找到的块，因为这个块即将被分配出去

块分割逻辑：
`if (page->property > n) {`检查找到的空闲块是否大于请求的大小，如果大于，需要进行块分割

`struct Page *p = page + n;`计算分割后剩余块的起始位置，page + n表示从当前块起始位置向后偏移n个页面

`p->property = page->property - n;`设置剩余块的大小为原大小减去分配的大小

`SetPageProperty(p);`标记剩余块的第一个页面具有PG_property标志，表示这是一个空闲块的起始

`list_add(prev, &(p->page_link));`将剩余的空闲块插入到链表中，使用之前保存的prev指针，插入到原位置，保持链表有序性

完成分配nr_free -= n;更新全局空闲页面计数，减去分配出去的n个页面

`ClearPageProperty(page);`清除分配出去的块的PG_property标志，因为这个块不再是一个空闲块，所以不需要这个标志

`return page;`返回分配的内存块起始页面指针

该函数用于分配给定大小的内存块。**如果剩余空闲内存块大小多于所需的内存区块大小，则从链表中查找大小超过所需大小的页，并更新该页剩余的大小。**

先查找第一个空闲块列表的块数量大于n的，赋值给page。然后会将对应的块分割成两部分，一部分用于分配，拿走了；另一部分保留在列表中。如果分裂后剩下的那块列表大小还是大于n的话，则更新剩余块的 property并将其添加到列表中。最后，减少 nr_free计数，并标记已分配的页面。

default_free_pages

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

```
static void
default_free_pages(struct Page *base, size_t n) {
```

函数声明：释放从base开始的n个连续物理页面

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

基本检查：确保释放的页面数量大于0

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

页面状态初始化循环：

`p = base`：遍历指针指向起始页面

`p != base + n`：遍历从base到base+n的所有页面

`assert(!PageReserved(p) && !PageProperty(p))`：**确保要释放的页面既不是保留页面也不是空闲页面（即当前应该是已分配状态）**

`p->flags = 0`：清空页面标志位

`set_page_ref(p, 0)`：设置页面引用计数为0

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

设置空闲块属性：

`base->property = n`：设置块首页面的property字段为释放的页面数量

`SetPageProperty(base)`：标记base页面具有PG_property标志，表示这是一个空闲块的起始

`nr_free += n`：更新全局空闲页面计数器

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

按地址顺序插入空闲链表：

`if (list_empty(&free_list))`：如果空闲链表为空，直接插入

`else`：链表不为空，需要找到正确的插入位置

`while`循环遍历链表寻找插入点：`le2page(le, page_link)`：链表节点转换为Page指针，`if (base < page)`：如果当前块的地址小于链表中的某个块，插入到该块之前，`else if (list_next(le) == &free_list)`：如果到达链表末尾，插入到链表尾部

```
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;
    }
}
```

向前合并（与前面的块合并）：

`le = list_prev(&(base->page_link))`：获取当前块的前一个节点

`if (le != &free_list)`：确保前一个节点不是链表头

`p = le2page(le, page_link)`：获取前一个块的Page指针

`if (p + p->property == base)`：检查前一个块的结束位置是否正好是当前块的开始位置`p->property += base->property`：合并块大小；`ClearPageProperty(base)`：清除原块的property标志；`list_del(&(base->page_link))`：从链表中删除原块，`base = p`：将base指向合并后的块起始位置

```
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));
    }
}
```

向后合并（与后面的块合并）：

`le = list_next(&(base->page_link))`：获取当前块的后一个节点

`if (le != &free_list)`：确保后一个节点不是链表头

`p = le2page(le, page_link)`：获取后一个块的Page指针

`if (base + base->property == p)`：检查当前块的结束位置是否正好是后一个块的开始位置；`base->property += p->property`：合并块大小；`ClearPageProperty(p)`：清除被合并块的property标志；`list_del(&(p->page_link))`：从链表中删除被合并的块

该函数用于释放内存块。将释放的内存块按照顺序插入到空闲内存块的链表中，并合并与之相邻且连续的空闲内存块。

如果该页面的保留属性和页面数量属性均不为初始值了，我们就重置页面的对应属性，将引用设置定义为0。然后对应的更新空闲块数的数量。然后，将页面添加到空闲块列表中，同时尝试合并相邻的空闲块。如果释放的页面与前一个页面或后一个页面相邻，会尝试将它们合并为一个更大的空闲块。

default_nr_free_pages

```
static size_t
default_nr_free_pages(void) {
    return nr_free;
}
```

该函数用于获取当前的空闲页面的数量。

basic_check

```
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);

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    assert(alloc_page() == NULL);
```

```
    free_page(p0);
    free_page(p1);
    free_page(p2);
    assert(nr_free == 3);

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

    assert(alloc_page() == NULL);

    free_page(p0);
    assert(!list_empty(&free_list));

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

    assert(nr_free == 0);
    free_list = free_list_store;
    nr_free = nr_free_store;

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

对前面的一些功能的检测，包括页面分配，引用计数，空闲页面的链接操作等。

default_check

```
static void
default_check(void) {
    int count = 0, total = 0;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p));
        count ++, total += p->property;
    }
    assert(total == nr_free_pages());

    basic_check();

    struct Page *p0 = alloc_pages(5), *p1, *p2;
    assert(p0 != NULL);
    assert(!PageProperty(p0));

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));
    assert(alloc_page() == NULL);
```

```
    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    free_pages(p0 + 2, 3);
    assert(alloc_pages(4) == NULL);
    assert(PageProperty(p0 + 2) && p0[2].property == 3);
    assert((p1 = alloc_pages(3)) != NULL);
    assert(alloc_page() == NULL);
    assert(p0 + 2 == p1);

    p2 = p0 + 1;
    free_page(p0);
    free_pages(p1, 3);
    assert(PageProperty(p0) && p0->property == 1);
    assert(PageProperty(p1) && p1->property == 3);

    assert((p0 = alloc_page()) == p2 - 1);
    free_page(p0);
    assert((p0 = alloc_pages(2)) == p2 + 1);

    free_pages(p0, 2);
    free_page(p2);

    assert((p0 = alloc_pages(5)) != NULL);
    assert(alloc_page() == NULL);

    assert(nr_free == 0);
    nr_free = nr_free_store;

    free_list = free_list_store;
    free_pages(p0, 5);

    le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        count --, total -= p->property;
    }
    assert(count == 0);
    assert(total == 0);
}
```

对内存管理系统进行更详细全面的检查，包括对空闲页面链表的遍历和属性检查、页面分配和释放的各种场景测试。

结构体default_pmm_manager

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

这个结构体用于内存管理相关的功能，其中包含了多个函数指针和一个字符串成员。以下是对各个成员的解释：

`.name = "default_pmm_manager"`：用于标识这个内存管理器的名称。

`.init = default_init`：函数指针，用于初始化内存管理器的某些状态。

`.init_memmap = default_init_memmap`：函数指针，用于设置内存页面的初始状态。

`.alloc_pages = default_alloc_pages`：函数指针，指向一个用于分配页面的函数。

`.free_pages = default_free_pages`：函数指针，指向一个用于释放页面的函数。

`.nr_free_pages = default_nr_free_pages`：函数指针，指向一个用于获取空闲页面数量的函数。

`.check = default_check`：函数指针，用于内存分配情况的检查。

物理内存分配过程

1. **初始化阶段**：系统启动时初始化空闲链表
2. **内存映射初始化**：将可用的物理内存区域初始化为空闲块
3. **分配请求**：
    - 遍历空闲链表，找到第一个大小足够的块
    - 如果块大小正好，直接分配
    - 如果块更大，分割并保留剩余部分
4. **释放过程**：
    - 将释放的块插入到正确位置（按地址排序）
    - 检查前后相邻块是否可以合并

各个函数的作用

详细作用和过程内容前面都已经分析过了，在此最后进行总结：

- default_init：**初始化空闲内存管理器，设置空闲页面数为0**
- default_init_memmap：**初始化内存映射，将一段连续的物理页面初始化为空闲块，并按照地址顺序插入到空闲链表中**

- default_alloc_pages：**分配物理页面，使用 first-fit 策略在空闲链表中查找第一个足够大的空闲块，进行分配并可能分割**
- default_free_pages：**释放物理页面，按地址顺序插入空闲链表，并尝试与相邻的空闲块合并**
- default_nr_free_pages：**获取当前空闲页面数量**
- basic_check：**基本功能检测**
- default_check：**进阶功能检测**
- 结构体default_pmm_manager：**方便后续调用，写成了结构体**

改进空间

1. 当前使用简单链表，可改成按块大小维护多个链表，提高查找效率
2. 搜索方法改进：使用二分法进行搜索空闲块
3. 分配方法可以再用更高效的方法
4. 定期合并小内存块
5. 维护常用大小的缓存块

设计实现特点

1. **地址有序链表**：空闲块按地址排序，便于合并操作
2. **块头信息**：每个空闲块的第一个页面记录整个块的大小
3. **双向链表**：便于插入和删除操作
4. **合并机制**：释放时自动合并相邻空闲块，减少碎片

练习二：实现 Best-Fit 连续物理内存分配算法(需要编程)

核心思想

要实现 `best-fit` 算法，就是需要在内存分配时选择最小的空闲块。而初始化 `free_list` 和初始化空闲块的方案，我们可以直接沿用 `first-fit` 算法的代码。

实现过程

首先，因为物理内存是唯一的，我们只需要一个 `page_list` 来管理真实的物理内存。因此，我们声明一个全局变量 `free_area` 用于管理空闲块。

首先需要初始化 `free_area` 的 `free_list` 和 `nr_free`，这部分的代码是写好的，直接沿用此代码：

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

接下来是初始化空闲块的代码，函数的基本框架已经提供，我们只需要完善一些细节即可。

首先是初始化页的时候，此时的页处于空闲态，因此我们需要把所有的`flag`写入`0`，并且不存在引用，所以页框的引用计数也设置为`0`。对于空闲块的第一个页，我们需要标注它的属性（也就是这个块包含的空闲页数）是`n`，而这个块里的其他页，属性都设置为0即可。对于第一页属性的设置，在`for`循环结束时我们进行了设置，因此在for循环内部，我们可以用`p->flags = p->property = 0;`统一设置标志和属性信息。

接下来需要把新申请的块插入到`free_list`中。因为`free_list`是按照空闲块的大小排序的，因此我们需要遍历`free_list`，找到第一个大于新申请的块的位置，将新申请的块插入到它前面。如果遍历到链表结尾都没有找到，那么就将新申请的块插入到链表尾部。这部分的代码和`first-fit`算法一致，我们可以参考沿用：

```
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: YOUR CODE*/
        // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
        p->flags = p->property = 0;
        set_page_ref(p, 0);

    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            /*LAB2 EXERCISE 2: YOUR CODE*/
            // 编写代码
            // 1、当base < page时，找到第一个大于base的页，将base插入到它前面，并退出循环
            // 2、当list_next(le) == &free_list时，若已经到达链表结尾，将base插入到链表
尾部
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }

        }
    }
}
```

接下来是`best_fit_alloc_pages()`函数，我们利用它来分配页，这也是和`first-fit`算法区别最大的部分。对于`best-fit`算法，在分配内存时，我们需要完整遍历`free_list`中所有的空闲块，然后选择符合要求且最小的空闲块。这段实现代码如下：

```
    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;
        }
    }
```

与`first-fit`不同，我们不存在找到第一个符合的块就`break`的跳出情况，而需要完整遍历整个`free_list`，对于每个空闲块，我们首先需要确认它的大小能满足我们的申请需求，也就是满足`(p->property >= n)`；其次，为了找到最小的符合情况的空闲块，我们需要比较符合要求的空闲块属性大小，如果当前空闲块的大小小于`min_size`，那么我们就更新`min_size`和`page`，也就是`(p->property < min_size)`。这里，我们初始化`min_size`为`nr_free + 1`，确保大于所有空闲块的大小。

当循环结束，如果找到了符合要求的空闲块，那么`page`就不会为`NULL`，接下来要做的操作就和`first-fit`一样了，我们需要取下申请大小的页，如果还有剩余，就把剩余的页重新链接回`free_list`。如果找不到符合要求的空闲块，那么直接返回`NULL`。这部分的代码也可以参考`first-fit`算法的代码：

```
    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;
```

最后是`best_fit_free_pages()`函数，这个函数用于回收我们释放的页。我们首先对释放的页的属性、标志位进行重新置位，然后设置好空闲块第一页的属性，再在整个`free_list`中找到正确的插入位置即可。如果新插入的空闲块和前面或后面的块直接相连，那么就要合并。这部分的代码也和`first-fit`算法一致，可以参考`first-fit`算法的代码：

```
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: YOUR CODE*/
    // 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: YOUR CODE*/
        // 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));
        }
    }
}
```

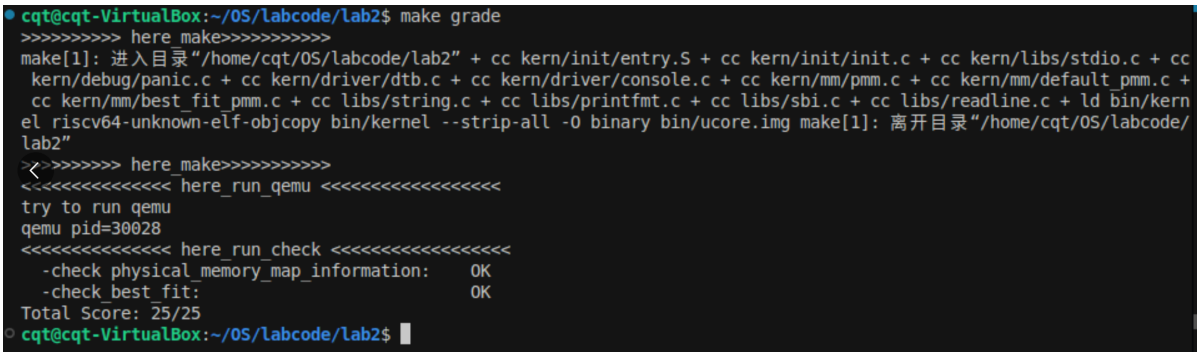
运行测试

我们对`best-fit`算法进行测试，修改`pmm.c`中的`pmm_manager`，使用我们新写好的`best_fit_pmm_manager`，然后进行`make qemu`测试，得到的测试结果如下图所示，显示我们成功通过所有测试：

```
cqt@cqt-VirtualBox:~/OS/labcode$ cd lab2
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make qemu
+ cc kern/mm/pmm.c
+ cc kern/mm/best_fit_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: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc0201660 (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc0205078 (virtual)
Kernel executable memory footprint: 20KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
```

```
lab2 > .score
Check PMM:                (2.3s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check ticks:                               OK
Total Score: 50/50
```

再进行make grade显示已经ok



Challenge1：buddy system(伙伴系统)分配算法(需要编程)

设计理念

Buddy System物理内存管理器旨在替代ucore原有的物理内存管理机制，通过二叉树结构管理物理内存页，提供高效的内存分配与释放能力，同时保持与ucore原有PMM接口的完全兼容。

核心思想

- **单位对齐**：Buddy System的最小管理单位与ucore物理内存管理的最小单位（页）保持一致
- **块页映射**：将物理页重新组织为"块"，1块 = 1页，便于与现有系统对齐
- **二叉树管理**：用完全二叉树结构替代原有的链表结构，提高分配效率
- **伙伴合并**：利用Buddy算法的特性，在释放时自动合并相邻空闲块，减少内存碎片

设计文档：详细设计信息说明

系统架构

1.数据结构设计

Buddy主结构体

```
struct buddy {
    unsigned size;          // 管理的总块数（必须是2的幂）
    unsigned longest[0];    // 柔性数组，记录子树中最大连续空闲块
};
```

设计说明：

- `size`：记录系统管理的总块数，必须为2的幂以保证二叉树完整性
- `longest[0]`：柔性数组，动态分配存储二叉树节点信息
- 二叉树节点数计算：对于n个块，需要2n-1个节点

内存布局

```
+-------------------+ <-- 内存起始地址
|   Buddy结构体      |
|   (管理数据结构)    |
+-------------------+ <-- buddy_page_base
|   可分配内存区域     |
|   (实际物理页)      |
+-------------------+
```

2. 关键宏定义

```
#define BUDDY_STRUCT_SIZE(size) (sizeof(struct buddy) + sizeof(unsigned) * (2 *
(size) - 2))
#define PAGES_TO_BLOCKS(pages) (pages)
#define BLOCKS_TO_PAGES(blocks) (blocks)
```

设计说明:

- `BUDDY_STRUCT_SIZE`：精确计算Buddy结构体所需内存大小
- 块与页的1:1映射简化了内存管理逻辑

核心算法实现

1.初始化过程

内存调整策略

```
// 步骤1：调整总页数为2的幂
total_pages = 1;
while (total_pages * 2 <= n) {
    total_pages *= 2;
}

// 步骤2：计算Buddy结构体占用页数
buddy_struct_pages = (buddy_struct_size + PGSIZE - 1) / PGSIZE;

// 步骤3：再次调整可管理页数为2的幂
size_t managed_pages = 1;
while (managed_pages * 2 <= available_pages) {
    managed_pages *= 2;
}
```

设计优势:

- 双重调整确保Buddy系统始终在2的幂次方内存上运行
- 自动适应不同大小的物理内存

二叉树初始化

```
unsigned node_size = blocks;
for (int i = 0; i < 2 * blocks - 1; ++i) {
    if (i == 0) {
        self->longest[i] = blocks;  // 根节点
    } else {
        if (is_pow_of_2(i + 1))
            node_size /= 2;
        self->longest[i] = node_size;
    }
}
```

2.内存分配算法

分配流程

1. **请求调整**：将请求页数调整为最近的2的幂
2. **容量检查**：检查根节点是否有足够空间
3. **节点搜索**：从根节点向下搜索合适大小的块
4. **分配标记**：标记选中节点为已分配
5. **回溯更新**：向上更新父节点的longest值

关键代码

```
// 寻找合适节点
for (; node_size != required_blocks; node_size /= 2) {
    if (self->longest[left_leaf(idx)] >= required_blocks)
        idx = left_leaf(idx);
    else
        idx = right_leaf(idx);
}

// 更新父节点
while (idx > 0) {
    idx = parent(idx);
    self->longest[idx] = max(self->longest[left_leaf(idx)],
                             self->longest[right_leaf(idx)]);
}
```

3.内存释放算法

释放流程

1. **参数验证**：检查释放参数的有效性
2. **节点定位**：根据块偏移找到对应的二叉树节点
3. **状态恢复**：将节点标记为空闲
4. **伙伴合并**：检查并合并相邻的空闲伙伴块
5. **回溯更新**：向上更新所有祖先节点

伙伴合并策略

```
while (idx > 0) {
    idx = parent(idx);
    node_size *= 2;
    unsigned left = self->longest[left_leaf(idx)];
    unsigned right = self->longest[right_leaf(idx)];
    if (left + right == node_size) {
        self->longest[idx] = node_size;  // 完全合并
    } else {
        self->longest[idx] = max(left, right);  // 部分合并
    }
}
```

与ucore PMM的集成

1.接口兼容性

```
const struct pmm_manager buddy_pmm_manager = {
    .name = "buddy_pmm_manager",
    .init = buddy_init,
    .init_memmap = buddy_init_memmap,
    .alloc_pages = buddy_alloc_pages,
    .free_pages = buddy_free_pages,
    .nr_free_pages = buddy_nr_free_pages,
    .check = buddy_check,
};
```

2.页面属性管理

- **分配时**：设置`PageProperty`标志，第一页记录分配页数
- **释放时**：清除页面引用和标志，恢复初始状态
- **兼容性**：完全支持ucore原有的页面管理机制

设计参考：

参考相关博客链接和仓库网站如下

https://coolshell.cn/articles/10427.html
https://github.com/wuwenbin/buddy2/blob/master/buddy2.c

BuddySystem设计优势

性能优势

1.时间复杂度

操作	时间复杂度	说明
分配	O(log n)	二叉树深度搜索
释放	O(log n)	二叉树回溯更新
查询	O(1)	直接访问根节点

2.空间效率

- **外部碎片**：通过伙伴合并显著减少
- **内部碎片**：由于2的幂次方分配，平均约25%
- **管理开销**：柔性数组精确计算，无额外浪费

架构优势

1. **无缝集成**：完全兼容ucore原有PMM接口
2. **高效管理**：二叉树结构提供快速分配
3. **自动优化**：伙伴合并减少内存碎片
4. **灵活扩展**：支持不同大小的物理内存

实现亮点

1. **精确计算**：柔性数组精确管理二叉树存储
2. **双重调整**：自动适配2的幂次方内存需求
3. **完整测试**：全面的功能验证和边界测试
4. **详细日志**：丰富的调试信息输出

测试结果

测试验证

1.测试策略

1. **基础功能测试**：单页、多页分配释放
2. **伙伴合并测试**：验证相邻块自动合并
3. **混合分配测试**：不同大小块交错分配
4. **边界情况测试**：零页、超限分配等
5. **完整性验证**：内存泄漏检查

2.测试覆盖

- 分配释放的正确性
- 伙伴合并的有效性
- 内存完整性保持
- 异常情况处理

详细信息见buddysystem测试输出信息

简易版输出信息：

```
cqt@cqt-VirtualBox:~/OS/labcode$ cd lab2
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make qemu
+ cc kern/mm/pmm.c
+ 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: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc0201396 (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc0205088 (virtual)
Kernel executable memory footprint: 20KB
memory management: buddy_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
buddy: initializing with 16384 pages (64 MB)
buddy: structure needs 131068 bytes (32 pages)
buddy: start of allocatable pages at 0xffffffffc020e7f0
buddy: available pages = 16352, start at 0xffffffffc020e7f0
buddy: memory initialization completed

=== Buddy System Check ===
Initial free pages: 16384
buddy_alloc: requesting 1 pages (1 blocks)
buddy_alloc: allocated 1 pages at block offset 0
single page allocation: 0xffffffffc020e7f0
buddy_alloc: requesting 2 pages (2 blocks)
buddy_alloc: allocated 2 pages at block offset 2
2 pages allocation: 0xffffffffc020e840
buddy_alloc: requesting 4 pages (4 blocks)
buddy_alloc: allocated 4 pages at block offset 4
4 pages allocation: 0xffffffffc020e890
Free pages after allocation: 8192
buddy_free: freeing 1 pages at block offset 0
freed 1 page
buddy_free: freeing 2 pages at block offset 2
freed 2 pages
buddy_free: freeing 4 pages at block offset 4
freed 4 pages
Free pages after free: 16384
buddy_check passed!
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
```

详细版本输出信息：

```
--- Test 1: Basic Allocation/Free ---
buddy_alloc: requesting 1 pages (1 blocks)
buddy_alloc: starting allocation for 1 blocks
buddy_alloc: root node has 8192 free blocks
buddy_alloc: starting search from root (idx=0, size=8192)
buddy_alloc: depth=1, current node idx=0, size=8192
buddy_alloc:   left child idx=1, free=4096
buddy_alloc:   right child idx=2, free=4096
buddy_alloc:   -> choosing LEFT child (idx=1)
buddy_alloc: depth=2, current node idx=1, size=4096
buddy_alloc:   left child idx=3, free=2048
buddy_alloc:   right child idx=4, free=2048
buddy_alloc:   -> choosing LEFT child (idx=3)
buddy_alloc: depth=3, current node idx=3, size=2048
buddy_alloc:   left child idx=7, free=1024
buddy_alloc:   right child idx=8, free=1024
buddy_alloc:   -> choosing LEFT child (idx=7)
buddy_alloc: depth=4, current node idx=7, size=1024
buddy_alloc:   left child idx=15, free=512
buddy_alloc:   right child idx=16, free=512
buddy_alloc:   -> choosing LEFT child (idx=15)
buddy_alloc: depth=5, current node idx=15, size=512
buddy_alloc:   left child idx=31, free=256
buddy_alloc:   right child idx=32, free=256
buddy_alloc:   -> choosing LEFT child (idx=31)
buddy_alloc: depth=6, current node idx=31, size=256
buddy_alloc:   left child idx=63, free=128
buddy_alloc:   right child idx=64, free=128
buddy_alloc:   -> choosing LEFT child (idx=63)
buddy_alloc: depth=7, current node idx=63, size=128
buddy_alloc:   left child idx=127, free=64
buddy_alloc:   right child idx=128, free=64
buddy_alloc:   -> choosing LEFT child (idx=127)
buddy_alloc: depth=8, current node idx=127, size=64
buddy_alloc:   left child idx=255, free=32
buddy_alloc:   right child idx=256, free=32
buddy_alloc:   -> choosing LEFT child (idx=255)
buddy_alloc: depth=9, current node idx=255, size=32
buddy_alloc:   left child idx=511, free=16
buddy_alloc:   right child idx=512, free=16
buddy_alloc:   -> choosing LEFT child (idx=511)
buddy_alloc: depth=10, current node idx=511, size=16
buddy_alloc:   left child idx=1023, free=8
buddy_alloc:   right child idx=1024, free=8
buddy_alloc:   -> choosing LEFT child (idx=1023)
buddy_alloc: depth=11, current node idx=1023, size=8
buddy_alloc:   left child idx=2047, free=4
buddy_alloc:   right child idx=2048, free=4
buddy_alloc:   -> choosing LEFT child (idx=2047)
buddy_alloc: depth=12, current node idx=2047, size=4
buddy_alloc:   left child idx=4095, free=2
buddy_alloc:   right child idx=4096, free=2
buddy_alloc:   -> choosing LEFT child (idx=4095)
buddy_alloc: depth=13, current node idx=4095, size=2
buddy_alloc:   left child idx=8191, free=1
buddy_alloc:   right child idx=8192, free=1
buddy_alloc:   -> choosing LEFT child (idx=8191)
buddy_alloc: found target node! idx=8191, size=1
buddy_alloc: allocated node idx=8191, block_offset=0
buddy_alloc: updating parent nodes...
```

```
buddy_alloc: allocated node idx=8191, block_offset=0
buddy_alloc: updating parent nodes...
buddy_alloc:   parent idx=4095, left=0, right=1, new_value=1
buddy_alloc:   parent idx=2047, left=1, right=2, new_value=2
buddy_alloc:   parent idx=1023, left=2, right=4, new_value=4
buddy_alloc:   parent idx=511, left=4, right=8, new_value=8
buddy_alloc:   parent idx=255, left=8, right=16, new_value=16
buddy_alloc:   parent idx=127, left=16, right=32, new_value=32
buddy_alloc:   parent idx=63, left=32, right=64, new_value=64
buddy_alloc:   parent idx=31, left=64, right=128, new_value=128
buddy_alloc:   parent idx=15, left=128, right=256, new_value=256
buddy_alloc:   parent idx=7, left=256, right=512, new_value=512
buddy_alloc:   parent idx=3, left=512, right=1024, new_value=1024
buddy_alloc:   parent idx=1, left=1024, right=2048, new_value=2048
buddy_alloc:   parent idx=0, left=2048, right=4096, new_value=4096
buddy_alloc: allocation completed! block_offset=0
buddy_alloc: allocated 1 pages at block offset 0
allocated 1 page at 0xffffffffc020f818
buddy_alloc: requesting 2 pages (2 blocks)
buddy_alloc: starting allocation for 2 blocks
buddy_alloc: root node has 4096 free blocks
```

```
buddy_free: freeing 4 pages at block offset 4
buddy_free: starting free process for offset=4, size=4
buddy_free: searching for target node from root (idx=0, size=8192)
buddy_free: depth=1, current node idx=0, size=4096, offset=4
buddy_free:   -> going LEFT (idx=1), offset remains 4
buddy_free: depth=2, current node idx=1, size=2048, offset=4
buddy_free:   -> going LEFT (idx=3), offset remains 4
buddy_free: depth=3, current node idx=3, size=1024, offset=4
buddy_free:   -> going LEFT (idx=7), offset remains 4
buddy_free: depth=4, current node idx=7, size=512, offset=4
buddy_free:   -> going LEFT (idx=15), offset remains 4
buddy_free: depth=5, current node idx=15, size=256, offset=4
buddy_free:   -> going LEFT (idx=31), offset remains 4
buddy_free: depth=6, current node idx=31, size=128, offset=4
buddy_free:   -> going LEFT (idx=63), offset remains 4
buddy_free: depth=7, current node idx=63, size=64, offset=4
buddy_free:   -> going LEFT (idx=127), offset remains 4
buddy_free: depth=8, current node idx=127, size=32, offset=4
buddy_free:   -> going LEFT (idx=255), offset remains 4
buddy_free: depth=9, current node idx=255, size=16, offset=4
buddy_free:   -> going LEFT (idx=511), offset remains 4
buddy_free: depth=10, current node idx=511, size=8, offset=4
buddy_free:   -> going LEFT (idx=1023), offset remains 4
buddy_free: depth=11, current node idx=1023, size=4, offset=4
buddy_free:   -> going RIGHT (idx=2048), new offset=0
buddy_free: found target node! idx=2048, size=4
buddy_free: set node idx=2048 to free (size=4)
buddy_free: starting buddy merging process...
buddy_free:   parent idx=1023, size=8
buddy_free:     left child idx=2047, free=4
buddy_free:     right child idx=2048, free=4
buddy_free:     -> MERGED! parent set to 8
buddy_free:   parent idx=511, size=16
buddy_free:     left child idx=1023, free=8
buddy_free:     right child idx=1024, free=8
buddy_free:     -> MERGED! parent set to 16
buddy_free:   parent idx=255, size=32
buddy_free:     left child idx=511, free=16
buddy_free:     right child idx=512, free=16
buddy_free:     -> MERGED! parent set to 32
buddy_free:   parent idx=127, size=64
buddy_free:     left child idx=255, free=32
buddy_free:     right child idx=256, free=32
buddy_free:     -> MERGED! parent set to 64
buddy_free:   parent idx=63, size=128
buddy_free:     left child idx=127, free=64
buddy_free:     right child idx=128, free=64
buddy_free:     -> MERGED! parent set to 128
buddy_free:   parent idx=31, size=256
buddy_free:     left child idx=63, free=128
buddy_free:     right child idx=64, free=128
buddy_free:     -> MERGED! parent set to 256
buddy_free:   parent idx=15, size=512
buddy_free:     left child idx=31, free=256
buddy_free:     right child idx=32, free=256
buddy_free:     -> MERGED! parent set to 512
buddy_free:   parent idx=7, size=1024
buddy_free:     left child idx=15, free=512
buddy_free:     right child idx=16, free=512
buddy_free:     -> MERGED! parent set to 1024
buddy_free:   parent idx=3, size=2048
buddy_free:     left child idx=7, free=1024
buddy_free:     right child idx=8, free=1024
buddy_free:     -> MERGED! parent set to 2048
buddy_free:   parent idx=1, size=4096
buddy_free:     left child idx=3, free=2048
buddy_free:     right child idx=4, free=2048
buddy_free:     -> MERGED! parent set to 4096
buddy_free:   parent idx=0, size=8192
buddy_free:     left child idx=1, free=4096
```

Challenge2：任意大小的内存单元slub分配算法(需要编程)

SLUB 原理概述

SLUB（The Simple List of Unused Blocks）是一种高效的内核对象分配器，主要用于频繁分配和释放小对象的场景。SLUB 分配器采用两层架构：

- **底层页分配器**：负责以页为单位分配和管理物理内存。SLUB 的底层直接复用默认的页分配算法（如 first-fit），无需额外设计。
- **上层对象分配器（SLUB层）**：在页分配器之上，将每个分配到的物理页切分为多个固定大小的小对象，并通过单链表管理所有空闲对象，实现高效的小对象分配和释放。

SLUB 核心思想

将一页内存切分为多个对象，每个对象的头部用于存储下一个空闲对象的地址，所有空闲对象串成单链表，分配和释放都只操作链表头，速度极快。

设计实现

主要数据结构

- **slub_slab_t**
  代表一个 slab（通常是一页），包含：
  - `list_entry_t list`：用于将 slab 挂到 partial/full 链表，方便 slab 管理和查找。[在2.2.2通过成员指针找结构体指针会有详细的介绍]
  - `void *free_list`：指向 slab 中空闲对象的单链表头。
  - `size_t free_count`：当前 slab 中空闲对象数量。
  - `struct Page *page`：该 slab 对应的物理页。
- **slub_cache_t**
  管理所有 slab，包含：
  - `size_t obj_size`：单个对象的大小。
  - `size_t objs_per_slab`：一个 slab 可容纳多少对象。
  - `list_entry_t slabs_partial`：部分空闲 slab 链表（有空闲对象可分配）。
  - `list_entry_t slabs_full`：已满 slab 链表（无空闲对象）。

关键实现方式

1.slab页初始化与头插法

每次分配到一页物理内存后，需将其切分为多个对象，并用“头插法”将所有对象串成单链表：

```
for (size_t i = 0; i < cache->objs_per_slab; i++) {
    void *obj = (char *)page_addr + i * cache->obj_size;
    *(void **)obj = slab->free_list;
    slab->free_list = obj;
}
```

解释：

- `obj` 是当前对象的地址。
- `*(void **)obj = slab->free_list;`：把原链表头写到新对象的头部（即“next”指针）。
- `slab->free_list = obj;`：更新链表头为新对象。
- 这样所有对象都串成一个单链表，链表头是 `slab->free_list`，分配和释放都只需操作链表头。

为了更好理解，先回顾一下指针的知识：

- `p` 是一个指针变量，比如 `int *p`，它存的是某个地址。
- `*p` 表示“指针指向的内容”，也就是**p指向的那块内存里的值**。
  - 举例：如果 `p` 指向一个整数变量 `a`，那么 `*p` 就是 `a` 的值。
- `&p` 表示“指针变量本身的地址”，也就是**p这个变量在内存中的位置**。

举例说明

```
int a = 10;
int *p = &a;
```

- `p` 的值是 `a` 的地址，比如 `0x1000`。
- `*p` 的值是 `a` 的内容，也就是 `10`。
- `&p` 是 `p` 这个指针变量自己的地址，比如 `0x2000`。

- obj` 是void*类型，里面存放的是一个地址，我们记为地址A
- `obj` 本身所在的地址，我们记为地址B
  [void *obj = (char *)page_addr + i * cache->obj_size;这句话是把地址B进行赋值]
- `(void **)obj` 代表把地址B当作一个指针
- `*(void **)obj` 代表把一个指针指向的内容读取出来，已知这个指针是地址B，地址B指向的内容是地址A，这句话就是把地址A读取出来
- 再之后就是把地址A赋值原本链表头节点，链表头节点指向这个新对象

2. 通过成员指针找结构体指针

SLUB 通过链表节点指针快速找到对应的 slab 结构体：

```
#define le2slub_slab(le, member) \
    ((slub_slab_t *)((char *)(le) - offsetof(slub_slab_t, member)))
```

解释：

- `le` 是结构体成员（如 `list`）的地址。
- `offsetof(slub_slab_t, member)` 得到该成员在结构体中的偏移。
- 用成员地址减去偏移，即可得到结构体起始地址，实现“成员指针反推结构体指针”。
- 这样可以通过链表节点快速定位到 slab 结构体，方便管理和操作。

3. slab中list成员的作用

- `list_entry_t list` 是为了将 slab 结构体挂到 partial/full 链表。
- 这样可以方便地遍历所有 slab，查找有空闲对象的 slab，或回收已满/空闲的 slab。
- 通过 `list`，结合 `le2slub_slab` 宏，可以高效地从链表节点定位到 slab 结构体，实现 slab 的动态管理。
- 所以其实list可以用slab中的任何成员代替，但是我们仍然设计出来list，是因为用专门的 list_entry_t list 更规范、更安全、更易维护

典型分配与释放流程

- **分配对象**：优先从 partial 链表的第一个 slab 分配对象，若无空闲 slab，则分配新页并初始化为 slab。
- **释放对象**：将对象头插回 slab 的空闲链表，若 slab 全部空闲，则归还物理页并从链表移除 slab。

测试结果

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

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)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc020138a (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc02050a8 (virtual)
Kernel executable memory footprint: 20KB
memory management: slub_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
end = 0xffffffffc02050a8
pages = 0xffffffffc0206000
npage = 557056, nbase = %zu
freemem = 0x0000000080346000
mem_begin = 0x0000000080346000, mem_end = 0x0000000088000000
init_memmap: base = 0xffffffffc020e2f0, n = %zu
finished page_init, npage = 557056
check_alloc_page() is running ...
==== SLUB OBJ CHECK BEGIN ====
slub_obj_alloc[0]=0xffffffffc0346fe0
slub_obj_alloc[1]=0xffffffffc0346fc0
slub_obj_alloc[2]=0xffffffffc0346fa0
slub_obj_alloc[3]=0xffffffffc0346f80
slub_obj_alloc[4]=0xffffffffc0346f60
slub_obj_alloc[5]=0xffffffffc0346f40
slub_obj_alloc[6]=0xffffffffc0346f20
slub_obj_alloc[7]=0xffffffffc0346f00
before free 8 objs, free obj count=120
when free 8 objs, free obj count=121
when free 8 objs, free obj count=122
when free 8 objs, free obj count=123
when free 8 objs, free obj count=124
when free 8 objs, free obj count=125
when free 8 objs, free obj count=126
```

```
when free 8 objs, free obj count=127
when free 8 objs, free obj count=0
after free 8 objs, free obj count=0
pages before=31929, pages after=31930
==== SLUB OBJ CHECK END ====
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
```

Challenge3：硬件的可用物理内存范围的获取方法(思考题)

通过硬件调试接口获取：

使用像JTAG硬件调试器直接连接到计算机的主板或处理器上。调试器可以以最高权限访问整个系统的物理地址空间，不受操作系统内存管理单元的限制。外部调试软件通过尝试读取不同地址并观察响应来判断某块内存区域是否存在。所有可被正常读取且内容稳定的区域通常被认为是可用的RAM，而对不存在的地址进行访问则会引发超时或错误，从而勾勒出可用的物理内存范围。

内存探测方法：

这种方法的思路是操作系统假设大部分内存都是可用的，逐块地访问内存，记录下可用内存的起止范围。从某个低地址开始，尝试向每个内存页进行写入和读取操作。如果某个地址能够成功写入并能正确读回，就认为该页面是可用RAM；如果操作导致机器卡死或异常，则认为该区域是不可访问的（可能是设备内存或根本不存在）。通过这种方式，可以大致勾勒出可用内存的边界。

然而，这种内存探测方法具有极大的破坏性和风险性。向未知的内存区域写入数据，极有可能覆盖正在运行的关键固件代码或导致硬件设备进入不可预测的状态，从而导致系统崩溃。因此，在实际的操作系统设计中，绝不会采用这种不可靠的方式。现代操作系统的标准做法是完全信任并利用Bootloader或固件传递过来的内存映射信息，并在此基础上安全地初始化自己的物理内存管理器。