Lab2

马浩祎 卢艺晗 陆皓喆

练习1：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。

实现过程

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。

代码分析

default_init

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

该函数用于初始化存放空闲块的链表，首先调用list_init函数，初始化一个空的双向链表free_list，然后定义了nr_free，也就是空闲块的个数定义为0。下面是对应的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); //判定n是否大于0，目的是确定我们需要存放的页面是否不为0，如果为0就不需要存放了。
    struct Page *p = base;//定义一个指针p，它的类型是指向Page结构体的指针，并将其初始化为指向base所指向的内存地址
    for (; p != base + n; p ++) {//通过for循环，遍历我们存放的每一个页面
        assert(PageReserved(p));//首先检查该页面是否为保留页面
        p->flags = p->property = 0;//将该页面的flags和property属性统统初始化为0
        set_page_ref(p, 0);//将页面的引用次数也记为0
    }
    base->property = n;//将首个页面的property属性设置为n，也就是对应的块总数
    SetPageProperty(base);//设置页面的属性值
    nr_free += n;//更新nr_free的数量
```

```
    if (list_empty(&free_list)) {//if函数用于判断该列表是否为空
        list_add(&free_list, &(base->page_link));//如果空的话，就将起始页面的链表节点添加到链表中
    } else {//如果不为空
        list_entry_t* le = &free_list;//首先初始化一个指针指向我们的空闲链表头
        while ((le = list_next(le)) != &free_list) {//一直往后找，找到了合适的位置，也就是地址大小按照顺序排列
            struct Page* page = le2page(le, page_link);
            if (base < page) {//如果新页面的起始地址base小于当前遍历到的页面page的地址，说明找到了合适的插入位置
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {//如果不是的话，就在后面链接即可
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

该函数用于初始化一个空闲内存块。参数base指向一个页面结构体数组的起始地址，代表一段连续的内存页面，也就是我们需要存放的数组；后面的参数n就是我们需要进行初始化的页面数量。

首先，我们判定n是否大于0，目的是确定我们需要存放的页面是否不为0，如果为0就不需要存放了。

然后，定义一个指针p，它的类型是指向Page结构体的指针，并将其初始化为指向base所指向的内存地址。然后，通过for循环，遍历我们存放的每一个页面。首先判定该页面是否为保留页面，如果是的话，就将该页面的flags和property属性统统初始化为0，同时将页面的引用次数也记为0。set_page_ref函数的功能是，将给定Page结构体指针所指向的页面的引用计数为指定的值。

```
static inline void set_page_ref(struct Page *page, int val) { page->ref = val; }
```

然后，将首个页面的property属性设置为n，也就是对应的块总数。然后设置页面的属性值。最后，更新nr_free的数量，进来几块空闲内存块，我们就加上对应的数量。

后面的if函数用于判断该列表是否为空：

- 如果空闲页面链表为空，则将起始页面的链表节点添加到链表中。
- 如果空闲页面链表不为空，则遍历链表找到合适的位置插入新的页面链表节点。具体是在链表中找到第一个地址大于base的页面，如果找到了就在该页面之前插入新的链表节点(使用list_add_before函数)；如果遍历完链表也没有找到这样的页面，则将新的链表节点添加到链表末尾(使用list_add函数)。

```
static inline void
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
}

static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}
```

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;
}
```

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。

首先，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));
        }
    }
}
```

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。

首先，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。

程序在进行物理内存分配的过程以及各个函数的作用

前面都已经分析过了，总结一下：

- default_init：**初始化空闲内存块的链表**，将空闲块的个数设置为0。
- default_init_memmap：**用于初始化一个空闲内存块。**先查询空闲内存块的链表，按照地址顺序插入到合适的位置，并将空闲内存块个数加n。

- default_alloc_pages：**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。
- default_free_pages：**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。
- default_nr_free_pages：**该函数用于获取当前的空闲页面的数量。**
- basic_check：**基本功能检测。**
- default_check：**进阶功能检测。**
- 结构体default_pmm_manager：方便后续的调用，写成了结构体。

改进空间

我认为有以下的方面可以进行相应的改进：

- 更高效的内存块合并策略，可以提升效率；
- 更快速的空闲块搜索算法，比如使用二分法进行搜索；
- 使用内存回收策略，提高内存的利用率；
- 更灵活的内存分配策略，比如后续会编写的一些内存分配算法。

练习2：实现 Best-Fit 连续物理内存分配算法(需要编程)

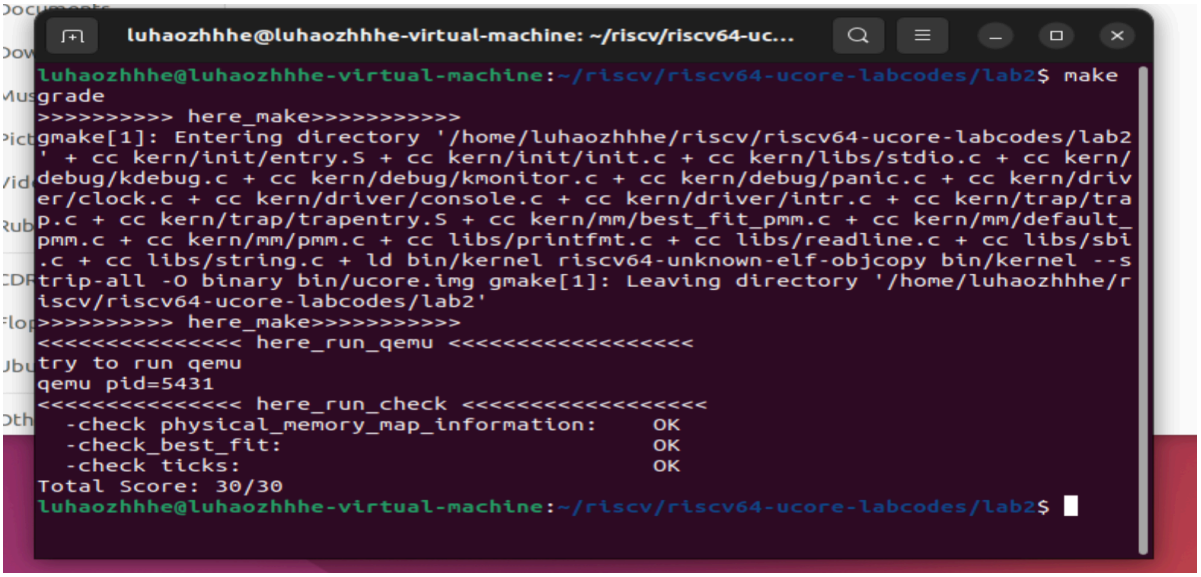
本部分需要编程的地方，只有一处和first-fit有所不同，其他地方我就不再赘述了，主要分析一下有关Best-Fit算法的代码部分：

```
 /*LAB2 EXERCISE 2: 2211044*/
// 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;
    }
}
```

我们设计的思路是：在分配内存块时，按照顺序查找，遇到第一块比所需内存块大的空闲内存块时，先将该块分配给page，之后继续查询，如果查询到大小比分配的内存块小的空闲内存块，将page更新为当前的内存块。释放内存块时，按照顺序将其插入链表中，并合并与之相邻且连续的空闲内存块。

我们在代码中，使用框架给出的min_size来记录最小块数的值，在循环过程中，如果有比当前的块大小要小的，就随时更新我们的min_size，直到循环结束，我们自然就知道了最小的块在哪，我们已经将最小块数的p值存储在了page变量中。

make qemu编译文件，然后make grade测试，得到以下的结果：



说明我们编写的程序是正确的！

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

主要思想

该问题我们主要需要解决的，无非是内存的分配和收回。其他的部分，我们的设计基本和best fit类似，只不过我们需要控制我们存储块存放的位置，不能将其放在一个链表上了。所以我们使用了数组链表的数据结构来解决存放空闲页表的问题。

首先设计分块的思想：我们把分配的内存块向上取到2的幂次的值，找到这个幂次大小的块对应的数组元素的下标，然后查看这个链表中有没有空闲的块，如果有的话就分配，没的话就往后遍历数组，去找到第一个含有空闲块的数组链表，然后使用递归分割操作完成分块。

然后是收回内存块的思想：首先，我们找到该内存块对应的数组下标的位置，然后检查是否有地址相邻的情况，如果有的话，就执行堆块合并操作；如果没有的话就直接存储即可。

主要的算法设计我将会在开发文档中详细说明。

开发文档

设计数据结构

由于我们要求每个存储块的大小必须是2的n次幂，所以我们需要设计一个新的数据结构，来存放我们对应的块。

我们参考原来的空闲链表的结构体设计，如下所示：

```
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // number of free pages in this free list
} free_area_t;
```

为了将buddy system中的可用存储空间按照2的幂次分别存储管理，我们尝试创建一个free_list数组free_array，设计新的数据结构如下所示：

```
typedef struct
{
    unsigned int max_order;                          // 实际最大块的大小
    list_entry_t free_array[15]; // 伙伴堆数组
    unsigned int nr_free;                            // 伙伴系统中剩余的空闲块
} buddy_system_t;
```

经过测试，发现在ucore系统中需要分配的内存块有31929个，所以实际上最大的情况就是存储16384页大小的链表，因此我们只需要设计到14次即可，最大值设置为15。

一些基础函数

一些程序中使用到的函数：

- IS_POWER_OF_2：判断是不是2的幂
- Get_Order_Of_2：返回一个数对应的幂指数
- Find_The_Small_2：找到该数对应的最接近的2的幂，比该数小
- Find_The_Big_2：找到该数对应的最接近的2的幂，比该数大
- show_buddy_array：显示指定范围内的空闲链表数组的内容，包括页数量和地址信息
- buddy_system_init：初始化buddy system
- buddy_system_init_memmap：初始化链表内容
- get_buddy：获取给定内存块地址的伙伴块地址
- buddy_system_nr_free_pages：输出当前的空闲页的数量

分块设计

设计了buddy_system_split函数用于处理空闲页表的分块操作。

```
static void buddy_system_split(size_t n){//分块操作

    assert(n>0&&n<=max_order);//判断n在不在数组的范围内
    assert(!list_empty(&(buddy_array[n])));//判断该列表是不是空的，只有不是空的情况下才能分裂

    struct Page *page1;//定义两个指向Page结构体的指针，用于分别表示即将分裂出的两个小块
    struct Page *page2;

    list_entry_t* le=list_next(&(buddy_array[n]));//获取对应块阶数为n的链表中的第一个元素的指针，list_next表示跳过链表头
    page1=le2page(le,page_link);//将链表元素指针转换为Page结构体指针
    size_t temp = 1;
    for(int i=0;i<n-1;i++){//计算分裂后的buddy块大小，应该是原来的一半
        temp=temp*2;
    }
    page2=page1+temp;//page2相当于是page1的buddy块，用page1的地址加上temp就变成了page2的地址

    page1->property=n-1;
```

```
    page2->property=n-1;//拆开之后，将两个分开的页表项的阶数属性都减一，因为大小减小了一半
    SetPageProperty(page1);
    SetPageProperty(page2);//更新完之后，更新两个新分裂后的页表的属性，包括flag等

    list_del(le);//操作完之后，把原来的大块删了
    list_add(&(buddy_array[n-1]),&page1->page_link);
    list_add(&(page1->page_link),&(page2->page_link));//在n-1的链表中，往后依次添加
page1和page2的链表项

    return;
}
```

首先我们输入一个对应的n值，也就是数组的下标位置。首先进行一些合法性判断，然后定义了两个Page指针，用于定义分裂的两个小块。然后获取到我们对应的数组位置的链表的第一个块，也就是需要被分割的块。然后我们将第一个分裂后的块地址不变，第二个块的地址紧跟在第一块的后面，也就是加上2的n-1次的地址。然后完成两个块的数组位置的下移，以及更新对应的属性，最后完成原始链表的删除和新链表的链接即可。

分配设计

```
static struct Page *
buddy_system_alloc_pages(size_t requested_pages)//用于完成内存块的分配
{
    assert(requested_pages > 0);//参数合法性检查

    if (requested_pages > nr_free)//如果请求的页大于最大页数，直接返回空值
    {
        return NULL;
    }

    struct Page *allocated_page = NULL;//初始化分配的页，为空
    size_t adjusted_pages = Find_The_Big_2(requested_pages); //首先，根据请求的页数，
找到我们对应需要去获取到页的数量，往上找
    size_t order_of_2 = Get_Order_Of_2(adjusted_pages);    //找到我们页数所在的数组位
置，也就是求出对应的幂指数

    bool found = 0;//初始化bool变量帮我们确定是否找到，然后进入while循环，直到找到为止
    while (!found)
    {
        if (!list_empty(&(buddy_array[order_of_2])))//查对应块阶数为order_of_2的链表
是否为空
        {
            allocated_page = le2page(list_next(&(buddy_array[order_of_2])),
page_link);//如果不空的话，就从对应的链表中取出第一个块，存储到allocated_page中
            list_del(list_next(&(buddy_array[order_of_2]))); //存完之后，就可以把原先
的那块空闲内存删了

            ClearPageProperty(allocated_page); //完成该页的属性更新
            found = 1;//同时将found置为1，代表已经找到了
        }
        else//如果为空的话，说明对应的那一个数组，没有对应的空闲块了，我们就要往后找
        {
            int i;
```

```
            for (i = order_of_2 + 1; i <= max_order; ++i)//该循环，找到从order_of_2
之后的第一个不为空的数组链表
            {
                if (!list_empty(&(buddy_array[i])))//只要不为空，就直接执行拆分操作
                {
                    buddy_system_split(i);//执行拆分操作
                    break;
                }
            }

            if (i > max_order)
            {
                break;
            }
        }
    }

    if (allocated_page != NULL)//完成剩余的nr_free的数量的更新
    {
        nr_free = nr_free - adjusted_pages;
    }

    return allocated_page;
}
```

传入一个参数为需要分配的页数大小。如果小于0或者大于上限，直接返回null或者中断程序。然后，我们根据请求的页数大小，找到比他大的第一个2的幂次数，也就是我们需要给他分配的页数大小。然后从其对应的数组位置开始遍历链表，找到第一个不为空的链表，即可进行块的分配。我们在此做了一个判断，如果刚好不需要拆分，也就是下标刚好对上，那就不用调用split函数；如果对应下标的链表为空的话，我们就往后寻找不为空的链表，然后使用循环进行递归拆分，直到链表不为空，停止。最后，我们更新当前空闲页面的数量，返回对应的分配页面即可。

释放设计

```
static void
buddy_system_free_pages(struct Page *base, size_t n)//完成内存的释放和内存块合并功能
{
    assert(n > 0);
    unsigned int p_number = 1 << (base->property); //计算出要释放的页的大小
    assert(Find_The_Big_2(n) == p_number);//确保要释放的页数量与块的大小匹配
    cprintf("Buddy System算法将释放第NO.%d页开始的共%d页\n", page2ppn(base),
p_number);
    struct Page *left_block = base; //初始化为传入的要释放的块的起始地址
    struct Page *buddy = NULL;
    struct Page *tmp = NULL;

    list_add(&(buddy_array[left_block->property]), &(left_block->page_link)); //
将当前块先插入我们对应的数组链表中

    buddy = get_buddy(left_block, left_block->property);//计算当前块的伙伴块地址，调用
函数并输出伙伴块的地址情况
    while (PageProperty(buddy) && left_block->property < max_order)//while循环来完
成伙伴块的递归合并操作
```

```
    {
        if (left_block > buddy)//如果当前块的地址大于伙伴块的地址，进入if分支，完成合并操
作
        {
            left_block->property = -1;
            SetPageProperty(base);

            tmp = left_block;
            left_block = buddy;
            buddy = tmp;//完成了buddy和left_block的交换操作，确保较小地址的块始终被标记为
left_block
        }

        list_del(&(left_block->page_link));
        list_del(&(buddy->page_link));//删除两个小块在链表中的位置
        left_block->property = left_block->property + 1; // 左快头页设置幂次加一，然
后将合并之后的数组位置再加一，完成合并操作

        list_add(&(buddy_array[left_block->property]), &(left_block->page_link));
// 头插入相应链表

        buddy = get_buddy(left_block, left_block->property);//再给出现在的页表地址情
况
    }
    SetPageProperty(left_block); //更新对应页表的属性
    nr_free += p_number;//操作完后，更新我们的空闲页表总数


    return;
}
```

首先我们计算出需要释放的页面的大小，也就是比他大的第一个2的幂指数。首先将left_block初始化为传入的要释放的块的起始地址，将当前块先插入我们对应的数组链表中。然后通过同等大小的页表块是否相邻来进行合并的操作。在合并的同时，我们需要同步更新页表对应的大小和位置，具体操作为：将合并后的地址进行交换，保持大小一致的顺序；删除原先两个小块在链表中的链接，更新其数组位置，然后链接在大块位置的数组上。最后，更新对应的空闲页表的数量。

测试样例

总述

```
static void
buddy_system_check(void){//我们的最终检测函数！！
    cprintf("BEGIN TO TEST OUR BUDDY SYSTEM!\n");
    buddy_system_check_easy_alloc_and_free_condition();
    buddy_system_check_min_alloc_and_free_condition();
    buddy_system_check_max_alloc_and_free_condition();
    buddy_system_check_difficult_alloc_and_free_condition();

}
```

我们完成了以下四方面的检测：

- 测试简单请求和释放操作
- 测试复杂请求和释放操作
- 测试请求和释放最小单元操作
- 测试请求和释放最大单元操作

测试简单请求和释放操作

```
static void
buddy_system_check_easy_alloc_and_free_condition(void)
{
    cprintf("CHECK OUR EASY ALLOC CONDITION:\n");
    cprintf("当前总的空闲块的数量为：%d\n", nr_free);
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;

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

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

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

    cprintf("p0的虚拟地址为:0x%016lx.\n", p0);

    cprintf("p1的虚拟地址为:0x%016lx.\n", p1);

    cprintf("p2的虚拟地址为:0x%016lx.\n", p2);

    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);

    cprintf("CHECK OUR EASY FREE CONDITION:\n");
    cprintf("释放p0...\n");
    free_pages(p0, 10);
    cprintf("释放p0后,总空闲块数目为:%d\n", nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p1...\n");
    free_pages(p1, 10);
    cprintf("释放p1后,总空闲块数目为:%d\n", nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
```

```
    cprintf("释放p2...\n");
    free_pages(p2, 10);
    cprintf("释放p2后,总空闲块数目为:%d\n", nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

}
```

该部分我们完成了简单分配和收回内存的功能检测。程序运行后，输出结果均正确，下面详细分析。

首先，原始的内存块的布局应该为：16384大小的一个块，如下所示：

```
BEGIN TO TEST OUR BUDDY SYSTEM!
CHECK OUR EASY ALLOC CONDITION:
当前总的空闲块的数量为：16384
```

然后，p0请求10页，程序执行buddy_system_alloc_pages函数，找到对应的下标为4，但是目前4号位的链表为空，于是往后遍历，遍历到的第一个不为空的链表即为16384的14号链表，于是执行split操作，将16384大小的内存块分割为下面的情况：

```
首先,p0请求10页
------------------当前空闲的链表数组:------------------
No.4的空闲链表有16页 【地址为0xffffffffc020f598】
No.5的空闲链表有32页 【地址为0xffffffffc020f818】
No.6的空闲链表有64页 【地址为0xffffffffc020fd18】
No.7的空闲链表有128页 【地址为0xffffffffc0210718】
No.8的空闲链表有256页 【地址为0xffffffffc0211b18】
No.9的空闲链表有512页 【地址为0xffffffffc0214318】
No.10的空闲链表有1024页 【地址为0xffffffffc0219318】
No.11的空闲链表有2048页 【地址为0xffffffffc0223318】
No.12的空闲链表有4096页 【地址为0xffffffffc0237318】
No.13的空闲链表有8192页 【地址为0xffffffffc025f318】
------------------显示完成!------------------
```

可以看到，初始的大内存块被分割为了一系列的小块，然后p0取走了我们的16页空闲页。

然后，p1再请求10页，相当于取走当前No.4的空闲链表中的16个空页：

```
然后,p1请求10页
------------------当前空闲的链表数组:------------------
No.5的空闲链表有32页 【地址为0xffffffffc020f818】
No.6的空闲链表有64页 【地址为0xffffffffc020fd18】
No.7的空闲链表有128页 【地址为0xffffffffc0210718】
No.8的空闲链表有256页 【地址为0xffffffffc0211b18】
No.9的空闲链表有512页 【地址为0xffffffffc0214318】
No.10的空闲链表有1024页 【地址为0xffffffffc0219318】
No.11的空闲链表有2048页 【地址为0xffffffffc0223318】
No.12的空闲链表有4096页 【地址为0xffffffffc0237318】
No.13的空闲链表有8192页 【地址为0xffffffffc025f318】
------------------显示完成!------------------
```

然后p2请求10页，相当于需要将No.5的链表中的空页拆分开来，输出如下结果：

```
最后,p2请求10页
------------------当前空闲的链表数组:------------------
No.4的空闲链表有16页 【地址为0xffffffffc020fa98】
No.6的空闲链表有64页 【地址为0xffffffffc020fd18】
No.7的空闲链表有128页 【地址为0xffffffffc0210718】
No.8的空闲链表有256页 【地址为0xffffffffc0211b18】
No.9的空闲链表有512页 【地址为0xffffffffc0214318】
No.10的空闲链表有1024页 【地址为0xffffffffc0219318】
No.11的空闲链表有2048页 【地址为0xffffffffc0223318】
No.12的空闲链表有4096页 【地址为0xffffffffc0237318】
No.13的空闲链表有8192页 【地址为0xffffffffc025f318】
------------------显示完成!------------------
```

然后开始我们的内存收回操作，调用buddy_system_free_pages函数，首先收回p0，也就是要收回16个空页，由于无地址相邻空页表，所以不执行合并，只完成收回。

```
CHECK OUR EASY FREE CONDITION:
释放p0...
Buddy System算法将释放第NO.525127页开始的共16页
释放p0后,总空闲块数目为:16352
------------------当前空闲的链表数组:------------------
No.4的空闲链表有16页 【地址为0xffffffffc020f318】
No.4的空闲链表有16页 【地址为0xffffffffc020fa98】
No.6的空闲链表有64页 【地址为0xffffffffc020fd18】
No.7的空闲链表有128页 【地址为0xffffffffc0210718】
No.8的空闲链表有256页 【地址为0xffffffffc0211b18】
No.9的空闲链表有512页 【地址为0xffffffffc0214318】
No.10的空闲链表有1024页 【地址为0xffffffffc0219318】
No.11的空闲链表有2048页 【地址为0xffffffffc0223318】
No.12的空闲链表有4096页 【地址为0xffffffffc0237318】
No.13的空闲链表有8192页 【地址为0xffffffffc025f318】
------------------显示完成!------------------
```

然后收回p1，由于地址与上一轮中的0xffffffffc020f318相邻，所以完成合并操作，同时调换链表地址顺序，保持大小关系。

```
释放p1...
Buddy System算法将释放第NO.525143页开始的共16页
释放p1后,总空闲块数目为:16368
------------------当前空闲的链表数组:------------------
No.4的空闲链表有16页 【地址为0xffffffffc020fa98】
No.5的空闲链表有32页 【地址为0xffffffffc020f318】
No.6的空闲链表有64页 【地址为0xffffffffc020fd18】
No.7的空闲链表有128页 【地址为0xffffffffc0210718】
No.8的空闲链表有256页 【地址为0xffffffffc0211b18】
No.9的空闲链表有512页 【地址为0xffffffffc0214318】
No.10的空闲链表有1024页 【地址为0xffffffffc0219318】
No.11的空闲链表有2048页 【地址为0xffffffffc0223318】
No.12的空闲链表有4096页 【地址为0xffffffffc0237318】
No.13的空闲链表有8192页 【地址为0xffffffffc025f318】
------------------显示完成!------------------
```

最后释放p2，地址相邻，我们循环做合并操作，最后还原到原来的内存状态，如下所示。

```
释放p2...
Buddy System算法将释放第NO.525159页开始的共16页
释放p2后,总空闲块数目为:16384
------------------当前空闲的链表数组:------------------
No.14的空闲链表有16384页 【地址为0xffffffffc020f318】
------------------显示完成!------------------
```

测试复杂请求和释放操作

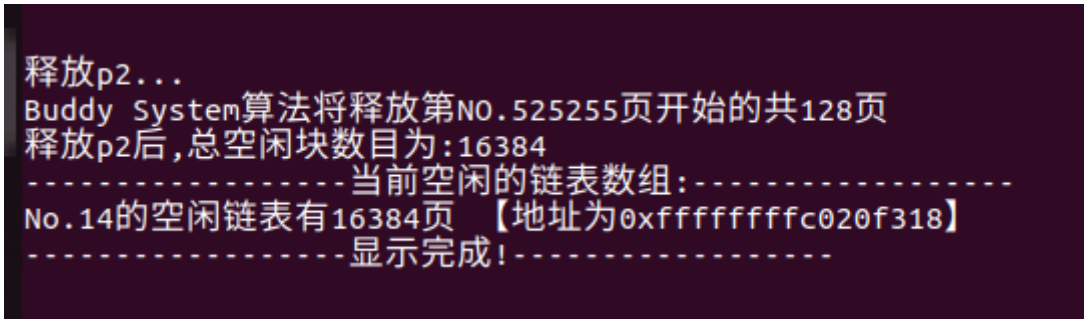
进阶版的请求和收回内存的测试样例，基本与上一个测试样例相同，由于篇幅原因，此处仅用语言描述一下。

首先，p0请求10页，也就是先将16384的块分割为8192,4096,2048,1024,512,256,128,64,32,16,16，然后取走16。

然后，p1请求50页，相当于将64直接拿走，内存情况变为8192,4096,2048,1024,512,256,128,32,16

最后，p2请求100页，相当于将128拿走，内存变为8192,4096,2048,1024,512,256,32,16

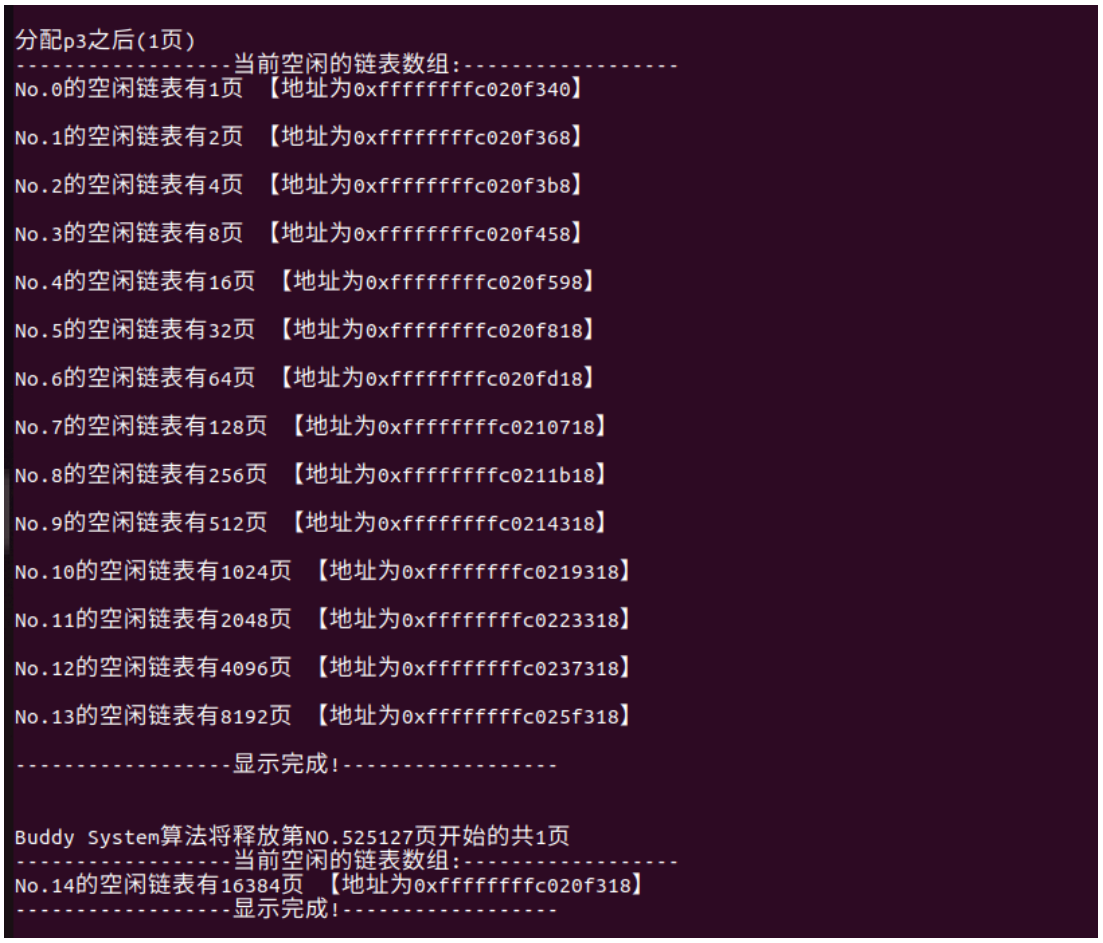
接着完成收回操作，此处不再赘述，给出最后的收回情况截图：



测试请求和释放最小单元操作

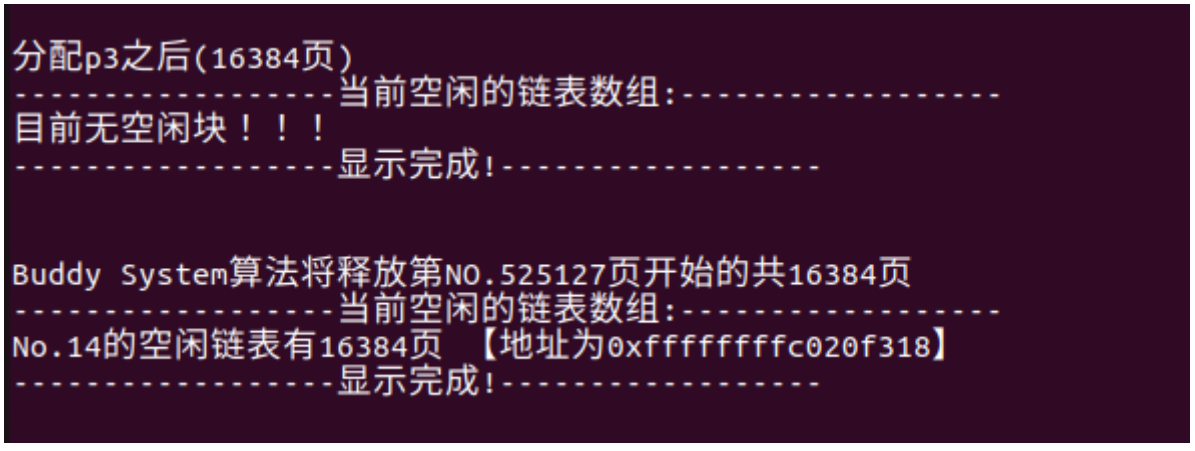
```c
static void buddy_system_check_min_alloc_and_free_condition(void){
    struct Page *p3 = alloc_pages(1);
    cprintf("分配p3之后(1页)\n");
    show_buddy_array(0, MAX_BUDDY_ORDER);

    // 全部回收
    free_pages(p3, 1);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

此处测试取走1个空闲块后的内存情况，和前面的情况类似，不再赘述，展示一下结果：



测试请求和释放最大单元操作

此处测试直接取走16384个空闲块后的内存情况，程序检测到全链表为空，输出“目前无空闲块！！！”



然后释放，内存又恢复原状了。

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（Slab Utilization By-pass）是 Linux 内核中的一种内存分配器，专门用于高效地管理内核中的小对象。它是 SLAB 分配器的改进版本，旨在提高性能、简化实现并减少内存碎片。SLUB 是现代 Linux 内核中默认的内存分配器，广泛用于分配和回收内核数据结构，如进程描述符、文件描述符等。

SLUB 核心思想

通过预分配固定大小的内存块（称为 slabs）来管理和分配内存。每个 slab 包含多个相同大小的对象，这些对象可以被快速分配和释放。

SLUB 主要机制

缓存（Caches）

- 缓存：SLUB 为每种大小的内存对象维护一个缓存（cache）。每个缓存对应一种特定大小的对象，管理着这些对象的分配和释放。
- 对象大小：每个缓存中的对象大小固定，这样可以减少内存碎片并提高分配效率。

Slab的管理

- Slab 结构：一个 slab 是一块连续的内存区域，包含多个相同大小的对象（在本实验中可以以页为单位）。每个 slab 都与一个特定的缓存相关联。
- 状态管理：每个 slab 可以处于三种状态之一：
    - 完全空闲（All free）：所有对象都未被分配。
    - 部分分配（Partial）：部分对象已被分配，部分对象仍然空闲。
    - 完全分配（Full）：所有对象都已被分配。

对象的分配和释放

- 分配对象：
    - 当需要分配一个对象时，SLUB 会在对应缓存的 slabs 中寻找第一个有空闲对象的 slab。
    - 如果找到一个部分分配的 slab，SLUB 会分配一个空闲对象并更新 slab 的状态。
    - 如果没有找到合适的 slab，SLUB 会创建一个新的 slab 并分配对象。
- 释放对象：
    - 当释放一个对象时，SLUB 会将其返回到所属的 slab，并更新 slab 的状态。
    - 如果一个 slab 中所有对象都被释放，SLUB 可以将该 slab 返回给内存池以供重用。

设计实现

我们在ucore中仿照SLUB的主要思想设计了简易版的实现，步骤如下：

1、设计思路

我们要实现的是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。
因此，我们保留页级别的分配策略，比如default（First-Fit）或Best-Fit算法，以当作第一层的分配，期望用户要求大内存分配的时候它们负责。而小内存需求由第二层来管理，这个内存需求应该小于4096KB（页大小），所以我将每个slab的占据单位设定为1页。
slab里存有固定大小的很多对象(可以通过观察检查部分的输出来理解)。关于slab的页面具体内容分配如下：

```
slab_struct_size  ||  obj  ||  bitmap
```

也就是说先存slab结构体大小，再存其中的很多对象，然后再存储表示每个obj位置是否被分配的位图。

2、新增数据结构Cache和Slab

```
typedef struct Slab{    //Slab 是一块连续的内存区域
                         //用于存储多个相同大小的对象
    list_entry_t list;  //链接自身的
    size_t free_cnt;    //空闲对象数量
    void *objs;         //对象寻址指针
    unsigned char *bitmap;//位图，标记对象的使用与否
}slab_t;
typedef struct Cache{
    list_entry_t slabs;//链接slab
    size_t obj_size;   //每个对象的大小
    size_t objs_num;   //一个slab有多少对象
}cache_t;
```

3、初始化

初始化分两层，我们先用默认页级别分配器进行页级别的初始化（后面slab的分配也是基于页的，统一按页初始化就比较方便）然后初始化cache数据结构：

```
static cache_t caches[3];//简化实现，3个cache
static size_t cache_n=0;//cache计数器
static void cache_init(void){
     cache_n=3;
     size_t sizes[3]={32,64,128};//对应大小
     for(int i=0;i<cache_n;i++){
         caches[i].obj_size=sizes[i];//大小设定
         caches[i].objs_num=calculate_objs_num(sizes[i]);//计算每个slab能存obj数量
         list_init(&caches[i].slabs);//初始化slab链表
     }
 }
```

Cache管理很多个Slab，Slab再存很多个obj，同Cache的obj大小都是一样的。
这里我们需要根据大小来计算数量，计算过程比较简单，我们设定x为个数，x是应满足如下不等式的最大整数：

$$slab\_struct\_size + x * size + (x+7)/8 <= 4096$$

很容易计算的，比如size为32KB的时候（slab结构体40KB），我们可以计算得x最大为126.
这样我们就初始化好两层的分配管理器了！

4、分配和释放

- 先根据size寻找最小的大于等于size的大小型号的Cache，找不到返回NULL
- 找到后进入链表项访问其中的Slab
- 如果该Slab已经满了，按链表访问下一个Slab，直到找到不满的Slab进行分配，分配时注意：返回对应obj指针，同时位图设置相应位，空闲数减一等操作。
- 如果都恰好满了，分配新的一页给新的Slab，并进行相应链接。

具体实现见代码。
对于释放，我们按照如下步骤进行

- 寻找该obj所在的Slab（遍历Cache，Slab，比较慢）
- 找到后直接进行相应释放操作
- 如果释放后整个Slab没有obj了，直接调用页级别的释放页面函数将其释放即可！

测试代码

以上就是我们根据Slub的主要思想设计的相应算法。下面我们给出测试我们机制的一些关键点，具体关键点的测试实现详见代码中的测试函数。

- 测试基本的分配/释放函数的功能（单个+多个）。
- 测试分配大小的边界条件。
- 测试大量分配释放的整体逻辑。
- 测试一些混合分配/释放流程。

Challenge3：硬件的可用物理内存范围的获取方法(思考题)

1. **手动探测物理内存**：一种操作上比较简单的方式是逐块地访问内存，记录下可用内存的起止范围。例如，逐个向内存块中写入测试数据，并读取；同时设计相应的异常捕获机制。当系统尝试访问超出物理内存范围的地址时，引发异常（如段错误等），OS需要能够捕捉到异常，以此来确定内存的边界。
2. **利用外设间接探测**：通过与外围设备交互来间接推测物理内存范围：可以通过配置DMA控制器，指定内存地址和数据长度，让DMA执行内存操作。若数据传输失败，则该内存地址可能无效或不存在。