buddy_pmm.h

仿照default_pmm.h，编写buddy_pmm.h

```
#ifndef __KERN_MM_BUDDY_PMM_H__
#define __KERN_MM_BUDDY_PMM_H__

#include <pmm.h>

extern const struct pmm_manager buddy_pmm_manager;
// 该内存分配管理 起名为buddy_pmm_manager

#endif /* ! __KERN_MM_BUDDY_PMM_H__ */
```

pmm.c

```
#include <buddy_pmm.h>
// 1.加上头文件
...

ppn_t first_ppn = 525127;
// 2.定义 第一个可分配的物理内存页在pages数组的下标

...

static void init_pmm_manager(void) {
    pmm_manager = &buddy_pmm_manager;
    // 我们要使用buddy_pmm_manager
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

pmm.h

添加下列代码

```
extern ppn_t first_ppn;
```

memlayout.h

仿照free_area_t，编写buddy system专用的结构体free_buddy_t。

```
#define MAX_BUDDY_ORDER 20

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;

typedef struct {
    list_entry_t free_array[MAX_BUDDY_ORDER + 1];
    // 可分配物理页链表数组，每个数组元素都一个free_list头
    unsigned int nr_free;
    // 剩余的空闲块，同free_area_t
    unsigned int max_order;
    // 伙伴二叉树的层数
} free_buddy_t;
```

buddy_pmm.c

仿照default_pmm.c，声明一个buddy system的基本结构，并定义一些量的别名，和上面的结构体的变量一一对应。

```
free_buddy_t buddy_s;

#define buddy_array (buddy_s.free_array)
#define max_order (buddy_s.max_order)
#define nr_free (buddy_s.nr_free)
```

定义一些辅助函数

```
// 判断是否为2的倍数
static size_t IS_POWER_OF_2(size_t n) {
    if (n & (n - 1))
        return 0;
        //位运算与操作可以很快识别是否为2的倍数
        //比如1000b，1000-1=111b，二者相与得到0
    else
        return 1;
}
// 获得 2^i < n < 2^(i+1) 的 i
static size_t getOrderOf2(size_t n) {
    size_t order = 0;
    while (n >> 1) {
        n >>= 1;
        order ++;
    }
```

```
    return order;
}
// 获得 2^i < n < 2^(i+1) 的 2^i
static size_t ROUNDDOWN2(size_t n) {
    size_t res = 1;
    if (!IS_POWER_OF_2(n)) {
        while (n) {
            n = n >> 1;
            res = res << 1;
        }
        return res>>1;
    }
    else {
        return n;
    }
}
// 获得 2^i < n < 2^(i+1) 的 2^(i+1)
static size_t ROUNDUP2(size_t n) {
    size_t res = 1;
    if (!IS_POWER_OF_2(n)) {
        while (n) {
            n = n >> 1;
            res = res << 1;
        }
        return res;
    }
    else {
        return n;
    }
}
```

定义函数用于展示buddy system当前的状态，用户测试输出。

```
static void
show_buddy_array(void) {
    cprintf("[TEST]Buddy System: Print buddy array:\n");
    cprintf("---------------------------\n");
    for (int i = 0;i < max_order + 1;i ++) {
        cprintf("No. %d: ", i);
        list_entry_t *le = &(buddy_array[i]);
        while ((le = list_next(le)) != &(buddy_array[i])) {
            struct Page *p = le2page(le, page_link);
            cprintf("%d ", p);
            cprintf("%d ", page2ppn(p));
            cprintf("%d ", 1 << (p->property));
        }
        cprintf("\n");
    }
    cprintf("---------------------------\n");
    return;
}
```

定义函数用于寻找buddy。

```
static struct Page*
buddy_get_buddy(struct Page *page) {
    size_t order = page->property;
    size_t buddy_ppn = first_ppn + ((1 << order) ^ (page2ppn(page) - first_ppn));
    // 1. page2ppn(page) - first_ppn代表虚拟内存的page在物理内存中和第一个可分配物理页的偏移值
    // 偏移值的分布应该是
    // 0: 1 2
    // 1: 1-2 3-4
    // 2: 1-4 5-8
    // 和1<<order进行异或，相当于是除了n的第order+1位取反，其他位都保持不变，因此可以算出buddy相对于first_ppn的偏移值
    cprintf("[TEST]Buddy System: Page NO.%d 's buddy page on order %d is: %d\n", page2ppn(page), order, buddy_ppn);
    if (buddy_ppn > page2ppn(page)) {
        //分情况讨论是+正偏移量还是-正偏移量以获得buddy的虚拟内存页地址
        return page + (buddy_ppn - page2ppn(page));
    }
    else {
        return page - (page2ppn(page) - buddy_ppn);
    }
}
```

仿照default_init_memmap定义初始化函数。

```
static void
buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    size_t pnum;
    size_t order;
    pnum = ROUNDDOWN2(n);
    // 将页数向下取整为2的幂
    order = getOrderOf2(pnum);
    // 求出页数对应的2的幂
    struct Page *p = base;
    // 初始化pages数组中范围内的每个Page
    for (; p != base + pnum; p ++) {
        assert(PageReserved(p));
        p->flags = 0;
        p->property = 0;
        // 初始化为非头页
        set_page_ref(p, 0);
        //↑和default里完全一样
    }
    max_order = order;
    nr_free = pnum;
    list_add(&(buddy_array[max_order]), &(base->page_link));
```

```
    // 将第一页base插入数组的最后一个链表，作为初始化的最大块16384的头页
    base->property = max_order;
    // 更新第一页的property（向下取2的幂）
    return;
}
```

定义函数用于分裂页块。

```
static void buddy_split(size_t n) {
    assert(n > 0 && n <= max_order);
    assert(!list_empty(&(buddy_array[n])));
    //检查有效值
    cprintf("[TEST]Buddy System: SPLIT!\n");
    struct Page *page_a;
    struct Page *page_b;
    //两个虚拟内存页指针

    page_a = le2page(list_next(&(buddy_array[n])), page_link);
    page_b = page_a + (1 << (n - 1));
    //确定两个指针的地址

    page_a->property = n - 1;
    page_b->property = n - 1;
    //因为分裂，各自的头指针都会减一

    cprintf("[TEST]Buddy System: a is %d ",page_a);
    cprintf("[TEST]Buddy System: b is %d ",page_b);

    list_del(list_next(&(buddy_array[n])));
    //从第n个free list中删除一块

    list_add(&(buddy_array[n-1]), &(page_a->page_link));
    list_add(&(page_a->page_link), &(page_b->page_link));
    //为第n-1个free list中添加两块
    return;
}
```

定义函数进行内存分配

```
static struct Page *
buddy_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    //↑和default_pmm.c一样
    size_t pnum = ROUNDUP2(n);
    // 所要分配的页数应该是n向上取整至2的幂次
```

```
    size_t order = getOrderOf2(pnum);
    // 求出所需页数对应的幂
    cprintf("[TEST]Buddy System: Allocating %d-->%d = 2^%d pages ...\n", n, pnum,
order);
    show_buddy_array();
    while(1)
    // 若存在对应的链表中含有空闲块，则直接分配
    if (!list_empty(&(buddy_array[order]))) {
        page = le2page(list_next(&(buddy_array[order])), page_link);
        list_del(list_next(&(buddy_array[order])));
        //因为已分配，所以从链表中删除
        SetPageProperty(page);
        // 将分配块的头页设置为已占用
        cprintf("[TEST]Buddy System: Buddy array after ALLOC NO.%d page:\n",
page2ppn(page));
        show_buddy_array();
        break;
    }
    // 若没有，则需要分裂
    else {
        for (int i = order+1;i < max_order + 1;i ++) {
            // 找到pow后第一个非空链表，分裂空闲块
            if (!list_empty(&(buddy_array[i]))) {
                buddy_split(i);
                cprintf("[TEST]Buddy System: Buddy array after SPLITT:\n");
                show_buddy_array();
                break;
            }
        }
    }
    nr_free -= pnum;
    // 可用数量减少
    cprintf("[TEST]Buddy System: nr_free: %d\n", nr_free);
    return page;
}
```

定义函数释放分配，并完成合并。

```
static void
buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    size_t pnum = 1 << (base->property);
    assert(ROUNDUP2(n) == pnum);
    cprintf("[TEST]Buddy System: Free NO.%d page about %d pages block: \n",
page2ppn(base), pnum);

    struct Page* left_block = base;
    //左块
    struct Page *buddy = NULL;
    struct Page* tmp = NULL;

    show_buddy_array();
```

```
    list_add(&(buddy_array[left_block->property]), &(left_block->page_link));
    // 先把左块释放，放回原列表
    cprintf("[TEST]Buddy System: add to list\n");
    show_buddy_array();

    buddy = buddy_get_buddy(left_block);
    // 获得当前块的buddy

    // 当伙伴块空闲，且当前块不为最大块时
    while (!PageProperty(buddy) && left_block->property < max_order) {
        cprintf("[TEST]Buddy System: Buddy free, MERGING!\n");
        if (left_block > buddy) {
            // 若当前左块为更大块的右块
            left_block->property = 0;
            ClearPageProperty(left_block);
            tmp = left_block;
            left_block = buddy;
            buddy = tmp;
        }
        list_del(&(left_block->page_link));
        // 删除左块
        list_del(&(buddy->page_link));
        // 删除右块
        left_block->property += 1;
        // 由于合并，property+1
        list_add(&(buddy_array[left_block->property]), &(left_block->page_link));
        // 重新插入更大一级的链表
        show_buddy_array();
        buddy = buddy_get_buddy(left_block);
    }
    cprintf("[TEST]Buddy System: Buddy array finished FREE:\n");
    ClearPageProperty(left_block);
    // 将回收块的头页设置为空闲
    nr_free += pnum;
    // 可以用页数增加
    show_buddy_array();
    cprintf("[TEST]Buddy System: nr_free is %d\n", nr_free);
    return;
}
```

定义测试函数。

```
static void
basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    cprintf("!!! 第一次分配 !!!\n");
    assert((p0 = alloc_page()) != NULL);
    cprintf("!!! 第二次分配 !!!\n");
    assert((p1 = alloc_page()) != NULL);
    cprintf("!!! 第三次分配 !!!\n");
    assert((p2 = alloc_page()) != NULL);
```

```
    cprintf("!!! 第一次释放 !!!\n");
    free_page(p0);
    cprintf("!!! 第二次释放 !!!\n");
    free_page(p1);
    cprintf("!!! 第三次释放 !!!\n");
    free_page(p2);
    show_buddy_array();

    cprintf("!!! 第四次分配-4 !!!\n");
    assert((p0 = alloc_pages(4)) != NULL);
    cprintf("!!! 第五次分配-2 !!!\n");
    assert((p1 = alloc_pages(2)) != NULL);
    cprintf("!!! 第六次分配-1 !!!\n");
    assert((p2 = alloc_pages(1)) != NULL);
    cprintf("!!! 第四次释放 !!!\n");
    free_pages(p0, 4);
    cprintf("!!! 第五次释放 !!!\n");
    free_pages(p1, 2);
    cprintf("!!! 第六次释放 !!!\n");
    free_pages(p2, 1);
    show_buddy_array();

    cprintf("!!! 第七次分配-3 !!!\n");
    assert((p0 = alloc_pages(3)) != NULL);
    cprintf("!!! 第八次分配-3 !!!\n");
    assert((p1 = alloc_pages(3)) != NULL);
    cprintf("!!! 第七次释放 !!!\n");
    free_pages(p0, 3);
    cprintf("!!! 第八次释放 !!!\n");
    free_pages(p1, 3);

    show_buddy_array();
    cprintf("!!! 测试结束 !!!\n");
}
```