ucore实验二：物理内存管理

笔记本：	My Notebook		
创建时间：	2022/7/14 15:37	**更新时间：**	2022/7/22 20:40
作者：	赵凌珂		
URL：	https://chyyuu.gitbooks.io/ucore_os_docs/content/lab2/lab2_3_2_1_phymemlab...		

作者：赵凌珂

练习1：实现 first-fit 连续物理内存分配算法

在实现first fit 内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。
提示:在建立空闲页块链表时，需要按照空闲页块起始地址来排序，形成一个有序的链表。可能会修改default_pmm.c中的default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数。请仔细查看和理解default_pmm.c中的注释。
请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 你的first fit算法是否有进一步的改进空间

实验过程与结果：

1.打开list.h：

```
struct list_entry {
    struct list_entry *prev, *next;
};
//list_entry是双链表

typedef struct list_entry list_entry_t;
//重命名

static inline void list_init(list_entry_t *elm) __attribute__((always_inline));
//初始化
static inline void list_add(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_add_before(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_add_after(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
//添加一个新的项目
static inline void list_del(list_entry_t *listelm)
__attribute__((always_inline));
```

```
//从列表中删除一个项目
static inline void list_del_init(list_entry_t *listelm)
__attribute__((always_inline));
//从列表中删除并重新定义
static inline bool list_empty(list_entry_t *list) __attribute__((always_inline));
//判断列表是否为空
static inline list_entry_t *list_next(list_entry_t *listelm)
__attribute__((always_inline));
//前一项
static inline list_entry_t *list_prev(list_entry_t *listelm)
__attribute__((always_inline));
//后一项


static inline void __list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t
*next) __attribute__((always_inline));
static inline void __list_del(list_entry_t *prev, list_entry_t *next)
__attribute__((always_inline));
```

2.打开memlayout.h可以查看物理页Page结构：

```
/* *
* struct Page - Page descriptor structures. Each Page describes one
* physical page. In kern/mm/pmm.h, you can find lots of useful functions
* that convert Page to other data types, such as phyical address.
* */
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status of
the page frame
    unsigned int property;          // the num of free block, used in first fit
pm manager
    list_entry_t page_link;         // free list link
};
```

3.打开default_pmm.c编写first-fit算法：

1）查看default_init函数：

```
free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

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

将free_area.free_list(全局的空闲链表入口)定义为free_list
将free_area.nr_free(全局空闲页数量)定义为nr_free
在default_init中，先将free_list进行init初始化，然后将空闲页数量置为0。

2）查看 default_init_memmap 函数：

```
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;
    list_add(&free_list, &(base->page_link));
}
```

3）查看default_alloc_pages 函数：

```
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_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            list_add(&free_list, &(p->page_link));
    }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

4）查看default_free_pages 函数：

```
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);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
```

```
        }
    }
    nr_free += n;
    list_add(&free_list, &(base->page_link));
}
```

5）修改函数：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);//当为假时中止程序
    struct Page *p = base;//声明一个base页
    for (; p != base + n; p ++) {//初始化
        assert(PageReserved(p));//检查
        p->flags = p->property = 0;//标记，property为0表示连续空页为0
        set_page_ref(p, 0);
    }
    base->property = n;//第一个页表base的property设置为n，表明有n个空闲块
    SetPageProperty(base);//标志位
    nr_free += n;//空闲页数目为n
    list_add_before(&free_list, &(base->page_link));//循环
}

static struct Page
default_alloc_pages(size_t n) {
    assert(n > 0);//n应该大于0
    if (n > nr_free) {
        return NULL;//无法分配
    }
    //有足够的空间分配
    struct Page *page = NULL;
    list_entry_t *le = &free_list;//声明链表头部
    while ((le = list_next(le)) != &free_list) {//遍历整个链表
        struct Page *p = le2page(le, page_link);//找到了合适的空闲空间
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) {//把多出的内存加到链表里
            struct Page *p = page + n;
            p->property = page->property - n;
            list_add_after(&free_list, &(p->page_link));
    }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}

static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {//初始化标记和ref
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;//将property改为n
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {检查能否合并
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
```

```
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    for(le = list_next(&free_list); le != &free_list; le = list_next(le))
    {
    p = le2page(le, page_link);
    if (base + base->property <= p) {
        assert(base + base->property != p);
        break;
    }
}
list_add_before(le, &(base->page_link));
}

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

练习二：实现寻找虚拟地址对应的页表项

通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的get_pte函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。本练习需要补全get_pte函数 in kern/mm/pmm.c，实现其功能。请仔细查看和理解get_pte函数中的注释。get_pte函数的调用关系图如下所示：

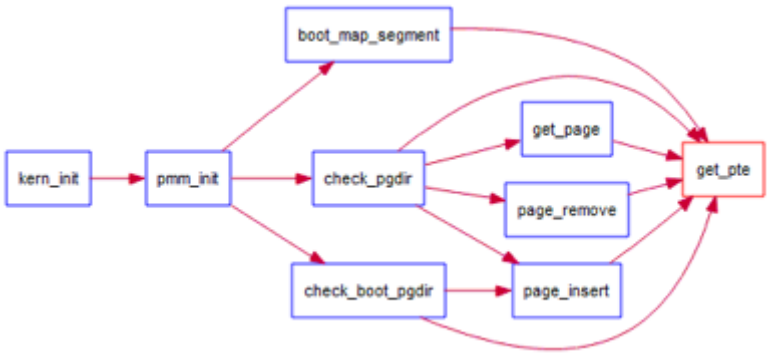

图1 get_pte函数的调用关系图

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处。

- 如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情？

实验过程和结果：

补全get_pte函数：

```
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
   //pgdir：PDT的内核虚拟地址；la：需要映射的线性地址；creat：决定是否分配页面；
   pde_t *pdep = &pgdir[PDX(la)];//一级页表
    if (*pdep & PTE_P) {//如果存在,返回ptep
        pte_t *ptep = (pte_t *)KADDR(*pdep & ~0x0fff) + PTX(la);
        return ptep;
    }
    //不存在分配page
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) {//如果不存在，
            return NULL;
    }
    set_page_ref(page, 1);//查找页面，引用次数+1
    uintptr_t pa = page2pa(page);//线性地址
    memset(KADDR(pa), 0, PGSIZE);//转成虚拟地址并初始化为0
    *pdep = pa | PTE_U | PTE_W | PTE_P;//设置
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];//返回ptep
}
```

练习三：释放某虚地址所在的页并取消对应二级页表项的映射

当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解page_remove_pte函数中的注释。为此，需要补全在 kern/mm/pmm.c中的page_remove_pte函数。page_remove_pte函数的调用关系图如下所示：

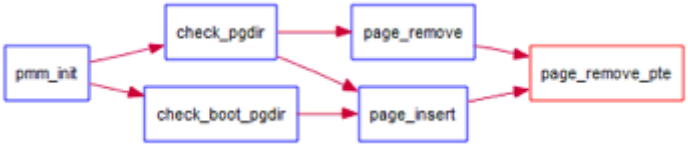

图2 page_remove_pte函数的调用关系图

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？
- 如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？鼓励通过编程来具体完成这个问题

实验过程和结果：

补全page_remove_pte函数：

```
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_P) {//检查page是否存在
        struct Page *page = pte2page(*ptep);
        if (page_ref_dec(page) == 0) {//如果ref为0，则free
            free_page(page);
        }
        *ptep = 0;//清除第二个页表ptep
        tlb_invalidate(pgdir, la);//刷新tlb
    }
}
```
