

实验3

个人信息

- 数据科学与计算机学院
- 2018级 软工3班
- 18342075
- 米家龙

目录

- 实验3
 - 个人信息
 - 目录
 - 实验名称
 - 实验目的
 - 实验内容
 - 实验要求
 - 实验环境
 - 实验过程
 - 练习0: 填写已有实验
 - 练习1: 实现 first-fit 连续物理内存分配算法
 - 查看 default_pmm.c 相关代码
 - list.h 中的相关数据结构和函数
 - 查看 memlayout.h 中的相关代码
 - 1. 修改 default_init() 函数
 - 2. 修改 default_init_memmap() 函数
 - 3. 修改 default_alloc_pages() 函数
 - 4. 修改 default_free_pages() 函数
 - 查看运行结果
 - 练习2: 实现寻找虚拟地址对应的页表项
 - 查看给出的注释
 - 补充 get_pte() 函数
 - 回答问题
 - 1. 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。
 - 2. 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?
 - 练习3: 释放某虚拟地址所在的页并取消对应二级页表项的映射
 - 实验结果

- 实验总结

- 完成实验后，请分析ucore_lab中提供的参考答案，请在实验报告中说明你的实现与参考答案的区别
 - 练习1
 - 1. default_init()
 - 2. default_init_memmap()
 - 3. default_alloc_pages()
 - 4. default_free_pages()
 - 练习2
 - 练习3
- 列出你认为本实验中重要的知识点，以及与其对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点
- 总结

实验名称

实验3 物理内存管理

实验目的

- 理解基于段页式内存地址的转换机制
- 理解页表的建立和使用方法
- 理解物理内存的管理方法

实验内容

- 了解如何发现系统中的物理内存
- 了解如何建立对物理内存的初步管理，即了解连续物理内存的管理
- 了解页表的相关操作，即如何建立页表实现虚拟内存到物理内存之间的映射，对段页内存管理机制有一个比较全面的了解

本实验里面实现的内存管理还是非常基本的，并没有涉及到对实际机器（如 cache）的优化

实验要求

- 练习0：填写已有实验
- 练习1：实现 first-fit 连续物理内存分配算法（需要编程）
- 练习2：实现寻找虚拟地址对应的页表项（需要编程）
- 练习3：释放某虚拟地址所在的页并取消对应二级页表项的映射（需要编程）

实验环境

使用老师提供的 `mooc-os-2015.vdi`，在虚拟机中创建 64 位的 Ubuntu 虚拟机并加载该 vdi，获得了版本为：

```
Linux moocos-VirtualBox 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014
x86_64 x86_64 x86_64 GNU/Linux
```

的虚拟机操作系统

并且使用 vscode 配合 Remote SSH 插件，实现通过远程终端在 windows 环境的对文件的编辑

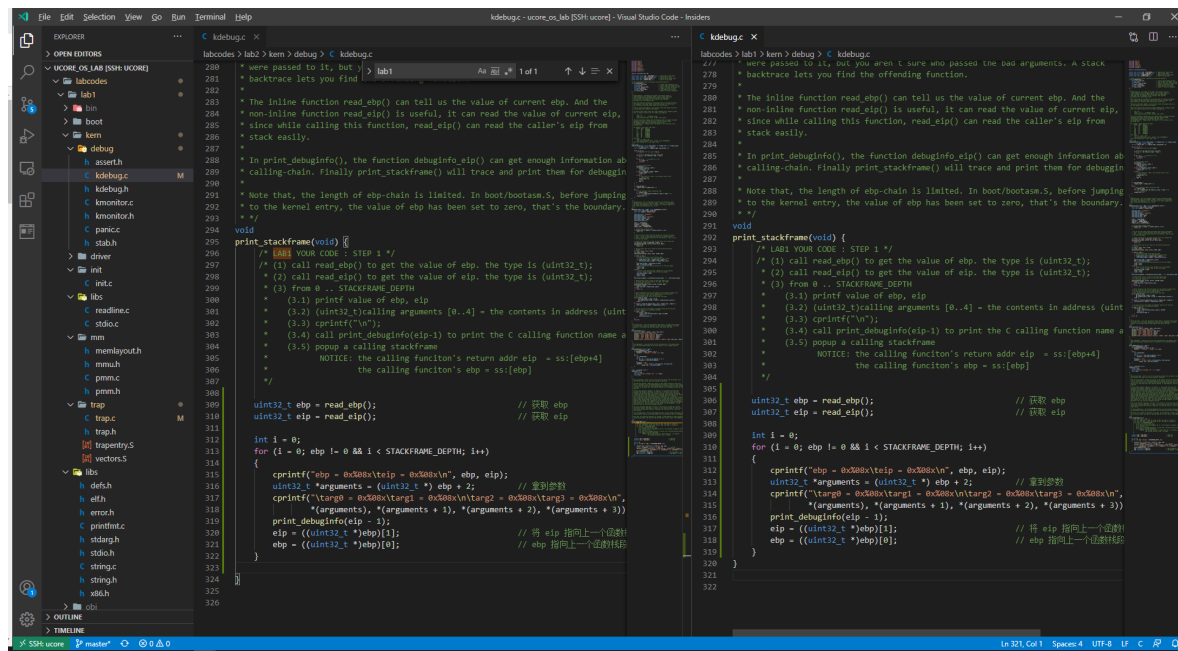
实验过程

练习0：填写已有实验

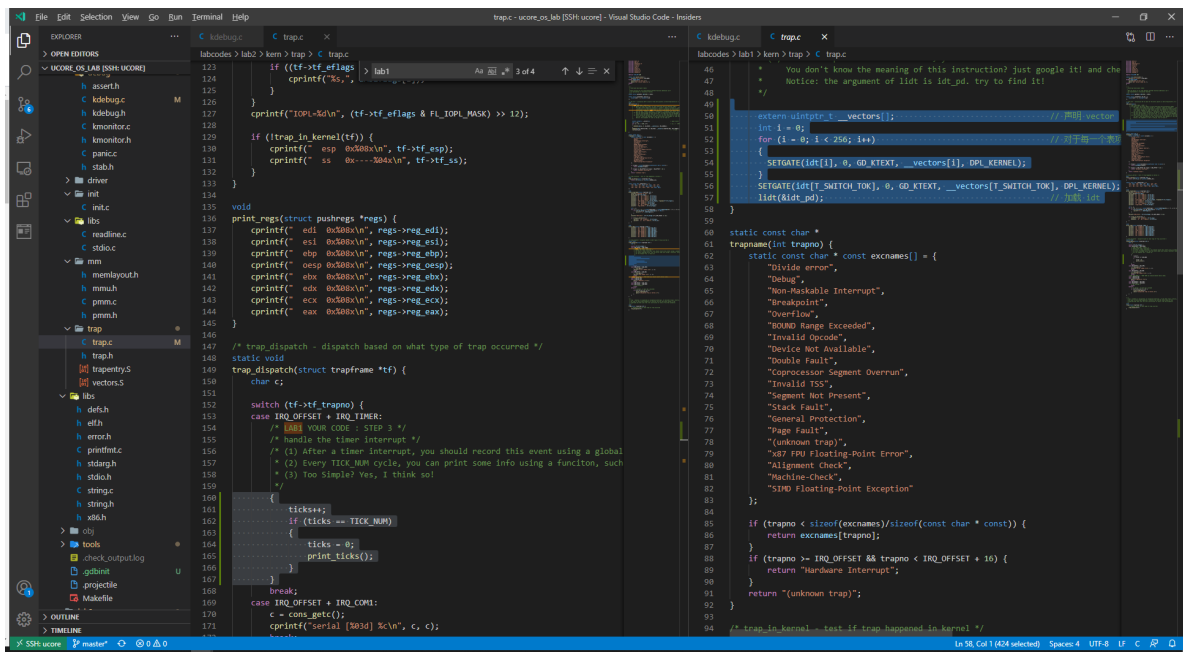
将实验2的代码填入本实验代码中有 `LAB1` 的注释相应部分

已知在实验2中被修改过的文件为 `kern/debug/kdebug.c` 和 `kern/trap/trap.c`，因此主要注意这两个文件即可

修改如图



The screenshot displays the Visual Studio Code interface with a Remote SSH connection to a Linux system. The Explorer pane on the left shows the file structure of the kernel, with `kern/debug` and `kern/trap` folders expanded. The main editor area shows the `kdebug.c` file, which contains several functions for debugging, including `print_stackframe` and `read_esp`. The code is written in C and includes comments in Chinese. The `print_stackframe` function is used to print the stack frame of the current function, and the `read_esp` function is used to read the stack pointer. The `trap.c` file is also visible in the Explorer pane, showing the `trap` function. The status bar at the bottom indicates the current file is `kdebug.c` and the cursor is at line 321, column 1.



练习1：实现 first-fit 连续物理内存分配算法

首次适应算法从空闲分区表的**第一个表目**起查找该表，把**最先能够满足要求**的空闲区分配给作业，这种方法目的在于**减少查找时间**。

- 优先利用内存**中低址部分**的空闲分区，从而保留了高址部分的大空闲区，为后续作业留下了大空间
- 减少了查找时间
- 会造成**外部碎片**

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，需要仔细查看和理解 default_pmm.c 的注释

查看 default_pmm.c 相关代码

查看注释，了解到需要修改如下函数

- default_init()
- default_init_memmap()
- default_alloc_pages()
- default_free_pages()

根据注释提示，查看其他相关代码

list.h 中的相关数据结构和函数

双链表 list_entry 数组结构，用于储存地址结构

```
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct list_entry list_entry_t; // 为什么要重命名，没有想明白
```

并存在这些函数

- list_init() 初始化一个 list_entry 双链表
- list_add() / list_add_after() 两者等价，将新元素插入到**列表元素与列表元素的下一个**之间
- list_add_before() 将新元素插入到**列表元素前一个和列表元素**之间

- list_del() 将传入的条目删除
- list_del() 从列表中删除一个条目并且重新初始化它
- list_empty() 判断列表是否为空
- list_next() 获取列表的最后一项
- list_prev() 获取列表的前一项
- __list_add() 列表增加的具体实现
- __list_del() 列表删除的具体实现

查看 memlayout.h 中的相关代码

先查看结构体 `Page` 的定义

```
struct Page {
    int ref; // 映射此物理页的虚拟页个数（该物理页被引用的次数）
    uint32_t flags; // 物理页的标志
    unsigned int property; // first-fit manager 的空闲块数量
    list_entry_t page_link; // 双向链表，用于连接各个页
};
```

查看 `le2page()` 函数

```
// 将列表转换成页
#define le2page(le, member) \
    to_struct((le), struct Page, member)
```

1. 修改 default_init() 函数

根据注释提醒的内容，需要

- 设定 `frea_list` 为0
- 设定 `nr_free` 为0

查看 `default_init()`，发现该函数可以直接使用

2. 修改 default_init_memmap() 函数

相关调用为：

1. kern_init
2. pmm_init
3. page_init
4. init_memmap
5. pmm_manager
6. init_memmap

该函数是用于**初始化空闲块**；而为了初始化一个块，则需要初始化该块的**每页**，具体操作为：

1. 设置 `p->flags` 中的 `PG_property` 位（可以参考 `pmm_init` 中设定 `p->flags` 的 `PG_reserved`）
 - 如果该页为空，并且不是空闲块的第一页，那么 `p->property` 应为0
 - 如果该页为空，是空闲块的第一页，那么 `p->property` 应为该块中**页的总数**
2. 设置 `p->ref` 为0，因为页是空闲并且无引用
3. 使用 `p->page_link()` 函数将该页与 `free_list` 链接起来
4. 更新空闲内存页的总数 `nr_free += n`

修改后为：

```

static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0); // 计算表达式, 如果结果为假, 则通过 stderr 打印出错信息, 然后调用 abort 中止
    程序
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(PageReserved(p)); // 检查是否为保留页, 是则中止
        p->flags = p->property = 0;
        SetPageProperty(p); // 设置为保留页
        set_page_ref(p, 0); // 设置 p->ref 为0
        list_add_before(&free_list, &(p->page_link)); // 插入空闲列表, 此处参照注释
    }
    base->property = n; // 这是第一页, 因此 property 需要设置为总页数
    // SetPageProperty(base); // 在循环中已经设置好了
    nr_free += n; // 空闲内存页的总数变化
    // list_add(&free_list, &(base->page_link));
}

```

3. 修改 default_alloc_pages() 函数

该函数是为了搜索 free_list 中的空闲的块（块大小大于等于 n），如果找到，则重新设置大小，并且返回块的地址，步骤如下：

1. 循环搜索检查 `p->property` 是否大于 n
 - 如果找到 `p`，意味着找到了空闲的、页数量大于等于 n 的块，设置 `PG_reserved` 为1，`PG_property` 为0，并且将这个页从链表中断开
 - 如果 `p->property > n` 那么我们需要从新计算剩下的页的数量
 - 重新计算 `nr_free` 的数量
 - 返回 `p`
2. 无法找到对应的块（块大小 $\geq n$ ），返回 `NULL`

修改后为

```

static struct Page *
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    // struct Page *page = NULL; // 用不到了
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) { // 遍历
        list_entry_t *tmp;
        struct Page *p = le2page(le, page_link); // 转换为页
        if (p->property >= n) { // 找到对应的页
            int i = 0;
            for (i; i < n; i++) // 将前 n 页从链表中断开
            {
                tmp = list_next(le);
                struct Page *page = le2page(le, page_link); // 转换成页
                SetPageReserved(page); // 将 PG_reserved 设置为1
                ClearPageProperty(page); // 将 PG_property 设置为0
                list_del(le); // 删除该节点
            }
            return p;
        }
    }
    return NULL;
}

```

```

        le = tmp;
    }
    if (p->property > n) // 需要分割页块
    {
        le2page(le, page_link)->property = p->property - n; // 分割页块
    }
    SetPageReserved(p); // 设置 p
    ClearPageProperty(p);
    nr_free -= n;
    return p;
}
}
return NULL;
}

```

4. 修改 default_free_pages() 函数

该函数作用在于重新将页链接回 free_list，或者将小的空闲块合并进大块里，步骤如下：

1. 根据块中的基址，在 free_list 中搜索到它正确的位置，然后插入到页中
2. 重置页的位置，相关参数为 `p->ref` 和 `p->flags`
3. 尝试合并块（需要正确修改 `p->property`）

思路为：遍历链表，从中找到第一页地址**大于**释放的第一页的地址

- 判断 *所释放的页的基地址 + 所释放的页的数量* 是否等于 *找到的地址*
 - 如果是，进行合并
- 找到该页之前的第一个 `p->property > 0` 的页，判断该页和上一页是否连续
 - 如果是，进行合并

修改后为：

```

static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base)); // 检查需要释放的页块是否已经被分配
    struct Page *tmp;
    list_entry_t *le = &free_list; // 用于寻找到释放位置
    while((le = list_next(le)) != &free_list) // 遍历，用于寻找到位置
    {
        tmp = le2page(le, page_link);
        if (tmp > base) break; // 找到
    }

    struct Page *p = base;
    for (; p != base + n; p++) {
        list_add_before(le, &(p->page_link)); // 从找到的位置开始，插入空闲页
    }
    base->flags = 0; // 重置参数
    base->property = n; // 设置块大小
    set_page_ref(base, 0); // 清空修改引用次数，设置为0
    ClearPageProperty(base); // 清空页 property，似乎可以不用这个
    SetPageProperty(base); // 再设置

    // 开始合并

    p = le2page(le, page_link);
    // 如果是高位，则向高位合并

```

```

// 假如 base + n = p , 那么决定向后合并
// 如果能够合并, 那么设置 base->property += p->property , 并将 p->property 设置为0
if (base + n == p)
{
    base->property += p->property;
    p->property = 0;
}
// 如果无法合并, 那么 p->property 本来就为0, 上述操作不产生影响

// 否则向前合并
// 向前合并需要: 如果前一页为空, 则之前所有都为空、可以合并的页
le = list_prev(&(amp;base->page_link));
p = le2page(le, page_link);
if(le != &free_list && tmp == base + 1) // 如果前一页为空闲页
{
    while (le != &free_list) { // 开始遍历
        if (p->property != 0)
        {
            p->property += base->property; // 更新该页的值
            base->property = 0;
            break;
        }
        le = list_prev(le); // 往前查询
        p = le2page(le, page_link);
    }
}
nr_free += n;
return;
}

```

查看运行结果


```
mijialong$>make qemu-nox
(THU.CST) os is loading ...
```

Special kernel symbols:

```
entry 0xc0100036 (phys)
etext 0xc0105d70 (phys)
edata 0xc011a000 (phys)
end 0xc011af28 (phys)
Kernel executable memory footprint: 108KB
ebp = 0xc0116f38 eip = 0xc01009dd
    arg0 = 0x00010094 arg1 = 0x00000000
    arg2 = 0xc0116f68 arg3 = 0xc01000c8
    kern/debug/kdebug.c:310: print_stackframe+22
ebp = 0xc0116f48 eip = 0xc0100cd0
    arg0 = 0x00000000 arg1 = 0x00000000
    arg2 = 0x00000000 arg3 = 0xc0116fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp = 0xc0116f68 eip = 0xc01000c8
    arg0 = 0x00000000 arg1 = 0xc0116f90
    arg2 = 0xffff0000 arg3 = 0xc0116f94
    kern/init/init.c:49: grade_backtrace2+33
ebp = 0xc0116f88 eip = 0xc01000f1
    arg0 = 0x00000000 arg1 = 0xffff0000
    arg2 = 0xc0116fb4 arg3 = 0x0000002a
    kern/init/init.c:54: grade_backtrace1+38
ebp = 0xc0116fa8 eip = 0xc010010f
    arg0 = 0x00000000 arg1 = 0xc0100036
    arg2 = 0xffff0000 arg3 = 0x0000001d
    kern/init/init.c:59: grade_backtrace0+23
ebp = 0xc0116fc8 eip = 0xc0100134
    arg0 = 0xc0105d9c arg1 = 0xc0105d80
    arg2 = 0x00000f28 arg3 = 0x00000000
    kern/init/init.c:64: grade_backtrace+34
ebp = 0xc0116ff8 eip = 0xc010008b
    arg0 = 0xc0105f60 arg1 = 0xc0105f68
    arg2 = 0xc0100c56 arg3 = 0xc0105f87
    kern/init/init.c:29: kern_init+84
```

memory management: default_pmm_manager

e820map:

```
memory: 0009fc00, [00000000, 0009fbff], type = 1.
memory: 00000400, [0009fc00, 0009ffff], type = 2.
memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07efe000, [00100000, 07ffdfff], type = 1.
memory: 00002000, [07ffe000, 07ffffff], type = 2.
memory: 00040000, [ffffc000, ffffffff], type = 2.
```

check_alloc_page() succeeded!

kernel panic at kern/mm/pmm.c:478:

```
    assertion failed: get_pte(boot_pgdir, PGSIZE, 0) == ptep
stack traceback:
```

```
ebp = 0xc0116f18 eip = 0xc01009dd
    arg0 = 0xc0106094 arg1 = 0xc0116f5c
    arg2 = 0x000001de arg3 = 0xc0103af8
    kern/debug/kdebug.c:310: print_stackframe+22
ebp = 0xc0116f48 eip = 0xc0100d41
    arg0 = 0xc0106964 arg1 = 0x000001de
    arg2 = 0xc0106989 arg3 = 0xc0106ae4
    kern/debug/panic.c:27: __panic+105
ebp = 0xc0116f88 eip = 0xc01047ff
    arg0 = 0x00000000 arg1 = 0xffff0000
    arg2 = 0xc0116fb4 arg3 = 0x0000002a
    kern/mm/pmm.c:478: check_pgdir+605
ebp = 0xc0116fc8 eip = 0xc010433c
```

```

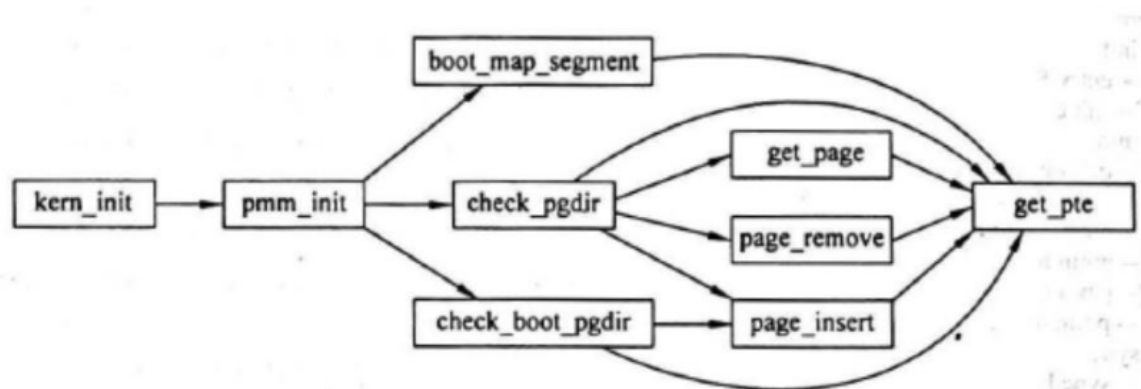
ebp = 0xc0110fc8      eip = 0xc010455c
    arg0 = 0xc0105d9c    arg1 = 0xc0105d80
    arg2 = 0x00000f28    arg3 = 0x00000000
    kern/mm/pmm.c:294: pmm_init+90
ebp = 0xc0116ff8      eip = 0xc0100090
    arg0 = 0xc0105f60    arg1 = 0xc0105f68
    arg2 = 0xc0100c56    arg3 = 0xc0105f87
    kern/init/init.c:31: kern_init+89
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
help - Display this list of commands.
kerninfo - Display information about the kernel.
backtrace - Print backtrace of stack frame.
Unknown command 'q'
Unknown command 'exit'
K> 

```

感觉自己代码逻辑比较混乱，暂时只能按照注释的提示来进行编写

练习2：实现寻找虚拟地址对应的页表项

从参考资料中得到 get_pte() 函数的调用关系图



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 kern/mm/pmm.c 中的 get_pte 函数，实现其功能。请仔细查看和理解 get_pte 函数中的注释。

查看给出的注释

在注释中，给出了一些宏和函数

- PDX(la) 返回虚拟地址 la 的页目录索引
- KADDR(pa) : 返回物理地址 pa 对应的虚拟地址
- set_page_ref(page,1) : 设置该页被引用1次
- page2pa(page): 得到 page 所管理的那一页的物理地址
- struct Page * alloc_page() : 分配一页
- memset(void *s, char c, size_t n) : 设置 s 指向地址的前 n 个字节的值为 c

补充 get_pte() 函数

通过查阅老师给的资料，阅读 `mmu.h` 中的注释，可以得到，32 位线性地址被分为3部分：

```

189
190 // A linear address 'la' has a three-part structure as follows:
191 //
192 // +-----10-----+-----10-----+-----12-----+
193 // | Page Directory | Page Table | Offset within Page |
194 // | Index | Index | |
195 // +-----+-----+-----+
196 // \--- PDX(la) --/ \--- PTX(la) --/ \--- PGOFF(la) ----/
197 // \----- PPN(la) -----/
198 //
199 // The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
200 // To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
201 // use PGADDR(PDX(la), PTX(la), PGOFF(la)).
202
203 // page directory index
204 #define PDX(la) (((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
205
206 // page table index
207 #define PTX(la) (((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF)
208
209 // page number field of address
210 #define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)
211
212 // offset in page
213 #define PGOFF(la) (((uintptr_t)(la)) & 0xFFF)
214
215 // construct linear address from indexes and offset
216 #define PGADDR(d, t, o) ((uintptr_t)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
217
218 // address in page table or page directory entry
219 #define PTE_ADDR(pte) ((uintptr_t)(pte) & ~0xFFF)
220 #define PDE_ADDR(pde) PTE_ADDR(pde)

```

- Directory , 一级页表, 存在高10位中, 可以通过 `PDX(la)` 获取
- Table , 二级页表, 存在中间10位中, 可以通过 `PTX(la)` 获取
- Offset , 偏移量, 存在低12位中, 可通过 `PGOFF(la)` 获取

从而可以发现, 每页大小为 $2^{12} = 4096 \text{ b}$, 即 4kb

补充 `get_pte()` 函数之后为:

```

pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    #if 0
        pde_t *pdep = NULL; // (1) find page directory entry
        if (0) { // (2) check if entry is not present
            // (3) check if creating is needed, then alloc page for page
            table
            // CAUTION: this page is used for page table, not for common
            data page
            // (4) set page reference
            uintptr_t pa = 0; // (5) get linear address of page
            // (6) clear page content using memset
            // (7) set page directory entry's permission
        }
        return NULL; // (8) return page table entry
    #endif
    // 上面的东西没有动

    pte_t *pdir = &pgdir[PDX(la)]; // 获取一级页表位置
    if (!((*pdir) & PTE_P)) // 如果不存在, 则需要判断是否需要创建二级页表
    {
        struct Page *p;

```

```

// 如果 create 为0, 则返回 NULL
// 如果 create 不为0, 则创建页表:
//                                     如果失败, 返回 NULL
//                                     如果创建成功, 返回对应二级页表的线性地址
if (create == 0 || (p = alloc_page()) == NULL) // 利用短路实现
    return NULL;
set_page_ref(p, 1); // 设置该页表引用次数 +1
uintptr_t pa = page2pa(p); // 得到物理地址
// 但是没有找到对应的函数, 只能先通过获取物理地址
再转化
memset(KADDR(pa), 0, PGSIZE); // 将物理地址转化为虚拟地址,
// 并且由于该页虚拟地址未被映射, 因此需要初始化

// *pdir = pa | PTE_P; // 设置存在
// *pdir = *pdir | PTE_U; // 设置可读
// *pdir = *pdir | PTE_W; // 设置可写
*pdir = pa | PTE_P | PTE_U | PTE_W; // 统一设置更加方便
}

return &((pte_t *)KADDR(PDE_ADDR(*pdir)))[PTX(la)]; // 得到二级页表对应的线性地址
}

```

回答问题

1. 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。

查阅 `mmu.h` 可以得到如下代码

```

/* page table/directory entry flags */
#define PTE_P      0x001 // Present
#define PTE_W      0x002 // Writeable
#define PTE_U      0x004 // User
#define PTE_PWT    0x008 // Write-Through
#define PTE_PCD    0x010 // Cache-Disable
#define PTE_A      0x020 // Accessed
#define PTE_D      0x040 // Dirty
#define PTE_PS     0x080 // Page Size
#define PTE_MBZ    0x180 // Bits must be zero
#define PTE_AVAIL  0xE00 // Available for software use
// The PTE_AVAIL bits aren't used by
the kernel or interpreted by the
// hardware, so user processes are
allowed to set them arbitrarily.

```

可以得到如下表格

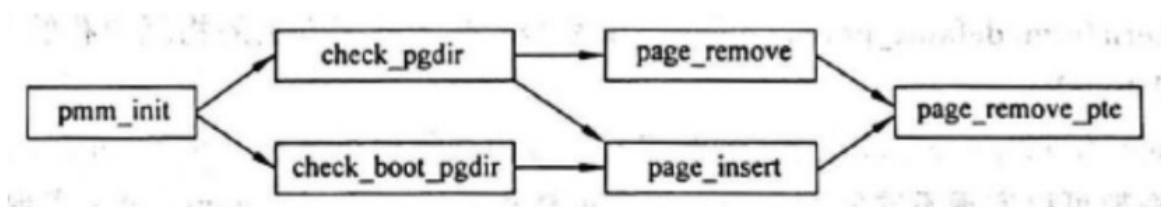
名称	地址所在位	作用
PTE_P	0	存在位
PTE_W	1	可写
PTE_U	2	访问该页所需要的特权级
PTE_PWT	3	是否使用 write-through
PTE_PCD	4	是否使用缓存, 1位不使用
PTE_A	5	是否被使用过
PTE_D	6	脏位
PTE_PS	7	页大小
PTE_MBZ	8	必须为0
PTE_AVAIL	9~11	内核和中断无效, 用户可以设置

2. 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?

1. 将引发异常的地址储存在 `cr2` 寄存器中
2. 判断特权级
 - 如果是内核态, 将 `EFLAGS`、`CS` 和 `EIP` 以及页访问代码 `error code` 依次压入 **中断栈** 中
 - 如果是用户态, 需要先压入 `ss` 和 `esp`, 在压入上面三种, 然后切换到内核态
3. 引发页错误, 根据 **中断描述符表** 查询并且跳转到对应页错误的 `ISR` 处执行, 然后将该页错误交给软件处理

练习3: 释放某虚拟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。



阅读注释, 可以了解到相关的宏的函数:

- `struct Page *page_pte2page(*ptep)`: 从 `ptep` 指针中得到对应的页
- `free_page`: 释放一个页
- `page_ref_dec(page)`: 减少引用次数, 但需要注意: 当次数为0时, 需要释放该页
- `tlb_invalidate(pde_t *pgdir, uintptr_t la)`: 使 TLB 的条目失效, 但该页表需处于处理器当前正在使用的页表中

补充后函数为:

```

//page_remove_pte - free an Page struct which is related linear address la
//                  - and clean(invalidate) pte which is related linear address la
//note: PT is changed, so the TLB need to be invalidate
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    #if 0
        if (0) {
            // (1) check if this page table entry is present
            struct Page *page = NULL; // (2) find corresponding page to pte
            // (3) decrease page reference
            // (4) and free this page when page reference reaches
            0
            // (5) clear second page table entry
            // (6) flush tlb
        }
    #endif

    if ((*ptep) & PTE_P) // 只有存在的时候才会进后续
    {
        struct Page *p = pte2page(*ptep); // 获取到页
        if (page_ref_dec(p) == 0) // 减少引用次数，同时判断次数是否为0
            free_page(p); // 如果只被上一级页表引用一次，那么可以直接释放页和对应的二级页表
        // 如果引用超过1次，那么不能释放页表，但是可以取消二级页表的映射
        // 无论是否释放页，都要取消二级页表的映射

        *ptep = 0;
        tlb_invalidate(pgdir, la);
    }

    // 如果不存在，什么都不干
    return;
}

```

实验结果

使用 `make qemu-nox` 在终端查看运行结果

```
mijialong$>make qemu-nox
(THU.CST) os is loading ...
```

Special kernel symbols:

```
entry 0xc0100036 (phys)
etext 0xc0105f0c (phys)
edata 0xc011a000 (phys)
end    0xc011af28 (phys)
```

Kernel executable memory footprint: 108KB

```
ebp = 0xc0116f38      eip = 0xc01009dd
      arg0 = 0x00010094      arg1 = 0x00000000
      arg2 = 0xc0116f68      arg3 = 0xc01000c8
      kern/debug/kdebug.c:310: print_stackframe+22
ebp = 0xc0116f48      eip = 0xc0100cd0
      arg0 = 0x00000000      arg1 = 0x00000000
      arg2 = 0x00000000      arg3 = 0xc0116fb8
      kern/debug/kmonitor.c:129: mon_backtrace+10
ebp = 0xc0116f68      eip = 0xc01000c8
      arg0 = 0x00000000      arg1 = 0xc0116f90
      arg2 = 0xffff0000      arg3 = 0xc0116f94
      kern/init/init.c:49: grade_backtrace2+33
ebp = 0xc0116f88      eip = 0xc01000f1
      arg0 = 0x00000000      arg1 = 0xffff0000
      arg2 = 0xc0116fb4      arg3 = 0x0000002a
      kern/init/init.c:54: grade_backtrace1+38
ebp = 0xc0116fa8      eip = 0xc010010f
      arg0 = 0x00000000      arg1 = 0xc0100036
      arg2 = 0xffff0000      arg3 = 0x0000001d
      kern/init/init.c:59: grade_backtrace0+23
ebp = 0xc0116fc8      eip = 0xc0100134
      arg0 = 0xc0105f3c      arg1 = 0xc0105f20
      arg2 = 0x00000f28      arg3 = 0x00000000
      kern/init/init.c:64: grade_backtrace+34
ebp = 0xc0116ff8      eip = 0xc010008b
      arg0 = 0xc0106100      arg1 = 0xc0106108
      arg2 = 0xc0100c56      arg3 = 0xc0106127
      kern/init/init.c:29: kern_init+84
```

memory management: default_pmm_manager

e820map:

```
memory: 0009fc00, [00000000, 0009fbff], type = 1.
memory: 00000400, [0009fc00, 0009ffff], type = 2.
memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07efe000, [00100000, 07ffdfdf], type = 1.
memory: 00002000, [07ffe000, 07ffffff], type = 2.
memory: 00040000, [fffc0000, ffffffff], type = 2.
```

check_alloc_page() succeeded!

check_pgdir() succeeded!

check_boot_pgdir() succeeded!

----- BEGIN -----

```
PDE(0e0) c0000000-f8000000 38000000 urw
|-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
```

----- END -----

++ setup timer interrupts

100 ticks

End of Test.

kernel panic at kern/trap/trap.c:18:

EOT: kernel seems ok.

stack traceback:

0xc0116f38: 0xc0116f38 0xc0100011


```

ebp = 0xc0116ee0      eip = 0xc01009dd
    arg0 = 0xc0106234      arg1 = 0xc0116f24
    arg2 = 0x00000012      arg3 = 0xc0116f5c
    kern/debug/kdebug.c:310: print_stackframe+22
ebp = 0xc0116f10      eip = 0xc0100d41
    arg0 = 0xc01062ee      arg1 = 0x00000012
    arg2 = 0xc01062d8      arg3 = 0x00000003
    kern/debug/panic.c:27: __panic+105
ebp = 0xc0116f30      eip = 0xc01018d4
    arg0 = 0xc0116f68      arg1 = 0xc0100343
    arg2 = 0xc01002f5      arg3 = 0xc0116f5c
    kern/trap/trap.c:18: print_ticks+65
ebp = 0xc0116f60      eip = 0xc0101d70
    arg0 = 0xc0116f8c      arg1 = 0xc0100366
    arg2 = 0xc0106264      arg3 = 0xc0116fa4
    kern/trap/trap.c:165: trap_dispatch+97
ebp = 0xc0116f80      eip = 0xc0101e2e
    arg0 = 0xc0116f8c      arg1 = 0x00000001
    arg2 = 0x00000000      arg3 = 0xc0116ff8
    kern/trap/trap.c:203: trap+16
ebp = 0xc0116ff8      eip = 0xc0101e46
    arg0 = 0xc0106100      arg1 = 0xc0106108
    arg2 = 0xc0100c56      arg3 = 0xc0106127
    kern/trap/trapentry.S:24: <unknown>+0
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> 

```

使用 `make grade` 进行测试，结果如下

```

mijialong$>make
+ cc kern/mm/pmm.c
kern/mm/pmm.c:265:1: warning: 'boot_alloc_page' defined but not used [-Wunused-function]
  boot_alloc_page(void) {
  ^
+ ld bin/kernel
+ ld bin/kernel_nopage
10000+0 records in
10000+0 records out
512000 bytes (5.1 MB) copied, 0.0381416 s, 134 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000356908 s, 1.4 MB/s
240+1 records in
240+1 records out
123260 bytes (123 kB) copied, 0.00104936 s, 117 MB/s
mijialong$>make grade
Check PMM: (2.6s)
  -check pmm: OK
  -check page table: OK
  -check ticks: OK
Total Score: 50/50
mijialong$>

```

实验总结

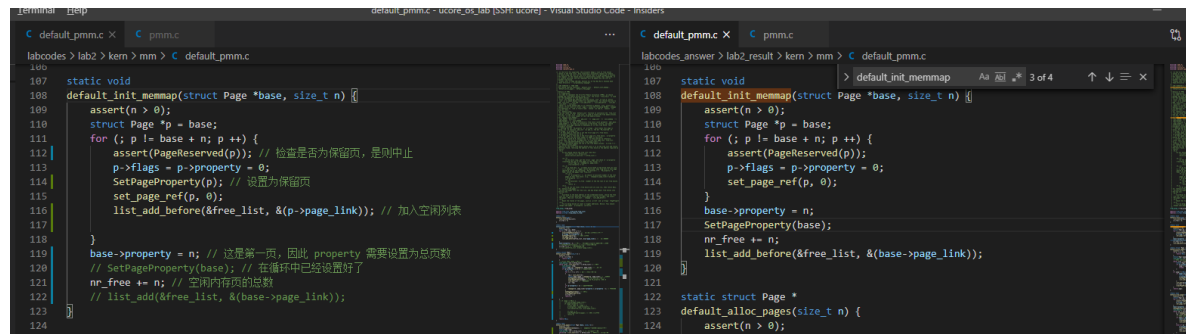
完成实验后，请分析ucore_lab中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别

练习1

1. default_init()

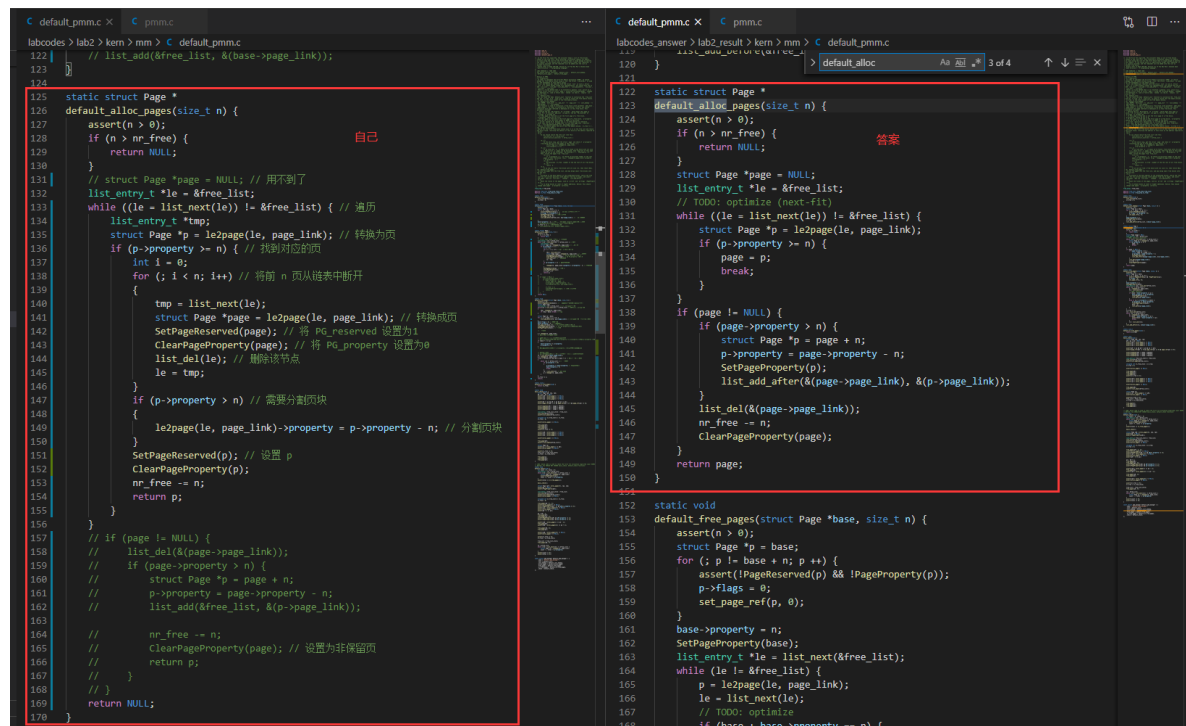
该函数不需要修改

2. default_init_memmap()



```
107 static void
108 default_init_memmap(struct Page *base, size_t n) {
109     assert(n > 0);
110     struct Page *p = base;
111     for (; p != base + n; p++) {
112         assert(PageReserved(p)); // 检查是否为保留页，是则中止
113         p->flags = p->property = 0;
114         SetPageProperty(p); // 设置为保留页
115         set_page_ref(p, 0);
116         list_add_before(&free_list, &(p->page_link)); // 加入空闲列表
117     }
118     base->property = n; // 这是第一页，因此 property 需要设置为总页数
119     // SetPageProperty(base); // 在循环中已经设置好了
120     nr_free += n; // 空闲内存的总数
121     // list_add(&free_list, &(base->page_link));
122 }
```

3. default_alloc_pages()



```
125 static struct Page *
126 default_alloc_pages(size_t n) {
127     assert(n > 0);
128     if (n > nr_free) {
129         return NULL;
130     }
131     // struct Page *page = NULL; // 用不到了
132     list_entry_t *le = &free_list;
133     while ((le = list_next(le)) != &free_list) { // 遍历
134         list_entry_t *tmp;
135         struct Page *p = le2page(le, page_link); // 转换为页
136         if (p->property >= n) { // 找到对应的页
137             int i = 0;
138             for (; i < n; i++) // 将前 n 页从列表中断开
139             {
140                 tmp = list_next(le);
141                 struct Page *page = le2page(le, page_link); // 转换成页
142                 SetPageReserved(page); // 将 PG_reserved 设置为1
143                 ClearPageProperty(page); // 将 PG_property 设置为0
144                 list_del(le); // 删除该节点
145                 le = tmp;
146             }
147             if (p->property > n) // 需要分割页块
148             {
149                 le2page(le, page_link)->property = p->property - n; // 分割页块
150             }
151             SetPageReserved(p); // 设置 p
152             ClearPageProperty(p);
153             nr_free -= n;
154             return p;
155         }
156     }
157     // if (page != NULL) {
158     //     list_del(&(page->page_link));
159     //     if (page->property > n) {
160     //         struct Page *p = page + n;
161     //         p->property = page->property - n;
162     //         list_add(&free_list, &(p->page_link));
163     //     }
164     //     nr_free -= n;
165     //     ClearPageProperty(page); // 设置为非保留页
166     //     return p;
167     // }
168     return NULL;
169 }
```

4. default_free_pages()

```
labcodes > lab2 > kern > mm > C default_pmm.c
175 | assert(PageReserved(base)); // 检查需要释放的页块是否已经被分配
176 | struct Page *tmp;
177 | list_entry_t *le = &free_list; // 用于寻找释放位置
178 | while(le != list_next(le)) != &free_list // 遍历, 用于寻找释放位置
179 | {
180 |     tmp = le2page(le, page_link);
181 |     if (tmp > base) break; // 找到
182 | }
183 |
184 | struct Page *p = base;
185 | for (; p != base + n; p++) {
186 |     list_add_before(le, &(p->page_link)); // 从找到的位置开始, 插入空闲页
187 | }
188 | base->flags = 0; // 重置参数
189 | base->xproperty = n; // 设置块大小
190 | set_page_ref(base, 0); // 清空修改引用次数, 设置为0
191 | ClearPageProperty(base); // 清空页 property, 似乎可以不用这个
192 | SetPageProperty(base); // 再设置
193 |
194 | // 开始合并
195 |
196 | p = le2page(le, page_link);
197 | // 如果是高位, 则向高位合并
198 | // 假如 base + n = p, 那么决定向后合并
199 | // 如果向前合并, 那么设置 base->xproperty += p->xproperty, 并将 p->xproperty 设置为0
200 | if (base + n == p)
201 | {
202 |     base->xproperty += p->xproperty;
203 |     p->xproperty = 0;
204 | }
205 | // 如果无法合并, 那么本条 p->xproperty 为0, 上述操作不产生影响
206 |
207 | // 否则向前合并
208 | // 向前合并需要: 如果前一页为空, 则之前所有都为空、可以合并的页
209 | le = list_prev(&(base->page_link));
210 | p = le2page(le, page_link);
211 | if (le != &free_list && tmp == base + 1) // 如果前一页为空闲页
212 | {
213 |     while (le != &free_list) { // 开始遍历
214 |         if (p->xproperty != 0)
215 |         {
216 |             p->xproperty += base->xproperty; // 更新该页的值
217 |             base->xproperty = 0;
218 |             break;
219 |         }
220 |         le = list_prev(le); // 往前查询
221 |         p = le2page(le, page_link);
222 |     }
223 | }
```

练习2

```
labcodes > lab2 > kern > mm > C pmm.c
358 | // (7) set page directory entry's permission
359 | return NULL; // (8) return page table entry
360 | #endif
361 | // 上面的东西没有动
362 | pte_t *pdir = &pgdir[PDX(la)]; // 获取一级页表位置
363 | if (!(*pdir & PTE_P)) // 如果不存在, 则需要判断是否需要创建二级页表
364 | {
365 |     struct Page *p;
366 |     // 如果 create 为0, 则返回 NULL
367 |     // 如果 create 不为0, 则创建页表
368 |     // 如果失败, 返回 NULL
369 |     // 如果成功, 返回二级页表的线性地址
370 |     if (create == 0 || (p = alloc_page()) == NULL) // 利用递归实现
371 |     {
372 |         return NULL;
373 |     }
374 |     set_page_ref(p, 1); // 如果要去查找页表, 则需要设置该页引用次数
375 |     uintptr_t pa = page2pa(p); // 得到物理地址
376 |     memset(KADDR(pa), 0, PGSIZE); // 但是没有找到对应的页表, 只能先通过获取物理地址
377 |     // 将物理地址转化为虚拟地址, 因此需要加
378 |     // *pdir = pa | PTE_P; // 设置存在
379 |     // *pdir = *pdir | PTE_U; // 设置可读
380 |     // *pdir = *pdir | PTE_M; // 设置可写
381 |     // *pdir = pa | PTE_P | PTE_U | PTE_M; // 统一设置更加方便
382 |     *pdir = pa | PTE_USER;
383 | }
384 | return &((pte_t *)KADDR(PDE_ADR(*pdir)))[PTX(la)]; // 得到二级页表对应的线性地址
385 |
386 |
387 |
388 |
389 | //page_remove_pte - free an Page struct which is related linear address la
390 |
```

练习3

```
labcodes > lab2 > kern > mm > C pmm.c
426 | // (3) decrease page reference
427 | // (4) and free this page when page reference reach 0
428 | // (5) clear second page table entry
429 | // (6) flush tlb
430 | #endif
431 |
432 | if ((*ptep & PTE_P) == 0) // 只有存在的时候才会进入后续
433 | {
434 |     struct Page *p = pte2page(*ptep); // 获取到页
435 |     if (page_ref_dec(p) == 0) // 减少引用次数, 同时判断次数是否为0
436 |     {
437 |         free_page(p); // 如果只被上一级页表引用一次, 那么可以直接释放页表和对应的二级页表
438 |         // 如果引用超过1次, 那么不能释放页表, 但是可以取消二级页表的映射
439 |         // 无论是清除页表, 都要取消二级页表的映射
440 |         *ptep = 0;
441 |         tlb_invalidate(pgdir, la);
442 |     }
443 | }
444 | // 如果不存在, 什么都不干
445 | return;
446 | }
447 |
448 | //page_remove - free an Page which is related linear address la and has an validate
449 | void
450 | page_remove(pte_t *pgdir, uintptr_t la) {
451 |     pte_t *ptep = get_pte(pgdir, la, 0);
452 |     if (ptep != NULL) {
453 |         page_remove_pte(pgdir, la, ptep);
454 |     }
455 | }
456 |
457 | //page_insert - build the map of phy addr of an Page with the linear addr la
458 | // parameters:
459 | // pdir: the kernel virtual base address of pdir
460 |
```

思路一致, 无区别

列出你认为本实验中重要的知识点，以及与其对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

本次实验涉及到

- CPU 段页式内存管理
- CPU 进入页机制的方法
- 连续物理内存分配方法 first-fit
- 虚拟地址（逻辑地址），线性地址和物理地址三者之间的关系和转换方法

对应到 os 原理中：

- 内存页管理
- 连续物理内存管理

理解：

- 实验中的知识点是原理中的具体应用，需要考虑到具体的操作，会更加的复杂繁琐
- 前者为后者提供具体的内存管理功能的详细操作和底层支持
- 前者涉及到了除开操作系统之外的一些知识，比如说数据结构和算法，使得实现起来稍微麻烦一些

列出你认为OS原理中很重要，但在实验中没有对应上的知识点

- 其他两种内存分配算法：best-fit 和 worst-fit，但是如果要实现这两种算法，花费时间可能会更长，并且一个文件的代码无法同时测试三种算法
- 虚拟内存的创建、实现和管理
- 出现了页错误时，软件具体如何处理
- 对于多线程/进程的创建、管理和调度，以及进程之间的互斥

总结

本次实验是从比较基本的 first-fit 算法开始进行实现，使我更加直观地了解了段页式内存的地址转换机制，页表及其条目的建立和使用，以及物理内存的管理；实现起来还是有点难度，而且工作量并不小，需要查看其他的文件的代码比较多，容易混淆；所幸每个练习的注释都十分的详细，按照注释的步骤和提示查阅资料，从而一步一步理解代码，最终实现了对应的功能，还是比较开心的。