《操作系统原理》lab3实验报告

专业：计算机科学与技术

姓名：郝裕玮

学号：18329015

1. 实验目的

①理解基于段页式内存地址的转换机制

②理解页表的建立和使用方法

③理解物理内存的管理方法

2. 实验过程与结果

练习 0：填写已有实验

本实验依赖实验 1。请把你做的实验 1 的代码填入本实验中代码中有“LAB1”的注释相应部分。提示：可采用 diff 和 patch 工具进行半自动的合并（merge），也可用一些图形化的比较/merge 工具来手动合并，比如 meld，eclipse 中的 diff/merge 工具，understand 中的 diff/merge 工具等。

答：工程量不大，需要修改的只有 debug/kdebug.c 和 trap/trap.c，手动复制修改即可。

练习 1：实现 first-fit 连续物理内存分配算法（需要编程）

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

答：在修改代码之前，需要对两个重要的数据结构进行解释：

①free_area_t：

```
/* free_area_t-维护一个双向链接列表以记录空闲（未使用）的页面 */
typedef struct {
    list_entry_t free_list;          // 链表头
    // 其中 list_entry_t 是一个将所有对象维护在一个双向链表中的数据类型
    unsigned int nr_free;            // 此空闲列表中的空闲页面数
} free_area_t;
```

所以该数据结构实际上是一个双向链表，负责管理所有的连续内存空闲块，便于分配和释放。

②Page：

```
/*
struct Page：页面描述符结构。每个页面描述一个物理页面。 在 kern / mm / pmm.h
中，您可以找到许多有用的函数，
这些函数可以将 Page 转换为其他数据类型，例如物理地址。
*/
struct Page {
    int ref;                         // 页面框架的参考计数器
    uint32_t flags;                  // 描述页面框架状态的标志数组
    unsigned int property;           // 空闲块的数量，在第一个
fit pm manager 中使用
    list_entry_t page_link;          // 空闲链表链接
};
```

1、ref：页被页表的引用记数，即映射此物理页的虚拟页个数。一旦某页表中有一个页表项设置了虚拟页到这个 Page 管理的物理页的映射关系，就会把 Page 的 ref 加 1。反之，若解除，则减 1。

2、flags：此物理页的状态标记，有两个标志位，第一个表示是否被保留，如果被保留了则设为 1（比如内核代码占用的空间）。第二个表示此页是否是 free（空闲）的。若为 1，则这页空闲，可以被分配；若为 0，则这页已经被分配出去了，不能再被二次分配。

3、property：记录某连续内存（mem）空闲块的大小。

4、page_link：把多个连续内存空闲块连接在一起的双向链表指针。连续内存空闲块利用这个页的成员变量 page_link 来连接比它地址小和大的其他连续内存空闲块。

PS：修改后代码中的中文注释为我自己补充的，英文注释则是复制了该文件开头对于我们补充代码的指导（复制英文注释表示我的该处代码是严格按照英文注释的指导写成的）

（1）对于 default_init( ):

修改前的代码为：

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

该函数作用为：初始化 free_list 及其头结点，并把 free_list 中的 nr_free 初始化为 0（即将当前空闲页数量设置为 0）。

在查阅上方对 default_init 的注释之后，可确定该函数的代码无需修改。

（2）对于 default_init_memmap ( ):

修改前的代码为：

```
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));
}
```

该函数作用为：初始化一个空闲块（带有参数：addr_base，page_number）。

结合注释可知，原代码仅仅把头部的空闲页 base 加入到了空闲链表中，但是中间遍历到的空闲页没有加入。同时，中间页的 flags 处未被设为 PG_property。

所以，修改后的代码为：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);//判断页数是否大于 0，若不大于 0 直接终止程序执行
    struct Page *p = base;//p 赋值为页头
    for (; p != base + n; p ++) {
        assert(PageReserved(p));//如果 assert 括号内的条件语句的值为 0，则直接终止程序执行
        //这里是判断 p 是否为保留页，即 flags 是否为 PG_reserved（是否为 0）
        p->flags = p->property = 0;//if this page is free and is not the first page of free block, p->property should be set to 0.
        SetPageProperty(p);//p->flags should be set bit PG_property (means this page is valid)
        set_page_ref(p, 0);//p->ref should be 0, because now p is free and no reference.
        list_add_before(&free_list, &(p->page_link));//We can use p->page_link to link this page to free_list, (such as: list_add_before(&free_list, &(p->page_link)); )
    }
    base->property = n;//if this page is free and is the first page of free block,p->property should be set to total num of block.
    nr_free += n;//Finally, we should sum the number of free mem block: nr_free+=n

}
```

（3）对于 default_alloc_pages ( ):

修改前的代码为：

```
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;
}
```

该函数作用为：在空闲链表中找到第一个空闲块（blocksize >= n），并调整空闲块的大小，最终返回被分配块的地址。

该代码的算法为：首先判断空闲页链表中的空闲页的个数是否大于 n，如果不大于则直接返回 NULL。
之后遍历整个空闲页链表，直到找到大于 n 的空闲块，若没找到则直接返回 NULL。
若找到，则需要将该空闲块先从空闲链表中删除。若该空闲块分配完 n 个页后仍有剩余，则将该块没分配完的部分重新加回到空闲链表中，并重新计算空闲页的个数，最后返回分配的页地址。

但原代码在处理链接多出来的空闲页时出现错误，应该是将找到的空闲块中被分配的 n 个空闲页从链表中删除。
再按照注释调用(le2page(le,page_link))->property=page->property-n 即可，而不是调用 list_add 使其直接与 free_list 链表头链接。

所以，修改后代码为：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);//判断页数是否大于 0，若不大于 0 直接终止程序执行
    if (n > nr_free) {
        return NULL;//若 n 超过空闲页数总量，则直接返回 NULL
    }
```

```
    struct Page *page = NULL;//page 是最后的返回值，初始化为 NULL
    list_entry_t *le = &free_list;//le 为空闲块链表头
    list_entry_t *next;//用于保存链表遍历到的当前节点
    int i;//同样用于链表遍历
    while ((le = list_next(le)) != &free_list) {//遍历链表，直到回到链表头
free_list
        struct Page *p = le2page(le, page_link);//将块地址转换成页地址
        if (p->property >= n) {//因为是 first-fit，所以遇到 property>n 的块
就选中
        //(4.1.2) If we find this p, then it' means we find a free bloc
k(block size >=n), and the first n pages can be malloced.
            page = p;//page 保存该空闲页地址
            break;
        }
    }
    if (page != NULL) {//若为 NULL 则证明遍历一遍链表都没有找到 property>n 的
块，此时 page 仍为 NULL，所以直接 return page
        struct Page* temp;//保存遍历时的当前空闲页
        for(i=1;i<=n;i++){//将需要分配的 n 个空闲页从链表中删除
            next=list_next(le);//链表节点向后移动，next 存储
            temp=le2page(le,page_link);//块地址转换成页地址
            SetPageReserved(temp);//设置当前页为保留页
            ClearPageProperty(temp);//清空当前页 Property
            //Some flag bits of this page should be setted: PG_reserved
 =1, PG_property =0
            list_del(le);//删除该节点
            //unlink the pages from free_list
            le=next;//节点移动，继续遍历
        }
        if (page->property > n) {//若该页块大小大于 n，即分配完之后还有剩余
的空闲页
            (le2page(le,page_link))->property = page->property - n;//因
为此时 le 指向分配过 n 个页后的那个页
            //所以将该页的 property 设置为 page->property-n（page 是分配该块
的首地址）
            //(4.1.2.1) If (p->property >n), we should re-
caluclate number of the the rest of this free block,
            //(such as: le2page(le,page_link))->property = p->property
- n;)
        }
        nr_free -= n;//空闲总页数-n
        //re-
caluclate nr_free (number of the the rest of all free block)
        ClearPageProperty(page);//清空当前页 property
```

```
        SetPageReserved(page);//设置当前页为保留页
    }
    return page;//返回 page（值为 NULL 或 p）
}
```

（4）对于 default_free_pages ( ):

修改前的代码为：

```
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));
}
```

该函数作用为：将释放的页重新链接回空闲页链表，同时也许会将小的空闲块合并为大的空闲块。

该代码的算法为：遍历链表，找到第一个页地址大于所释放的第一页的地址（一般是紧随的）。同时判断所释放的页的基地址加上所释放的页的数目是否刚好等于找到的地址，如果是则进行合并。

同理我们需要找到这个页之前的第一个 property 不为 0 的页，并判断所释放的页和上一个页是否是连续的，如果连续则进行合并操作。

所以，修改后代码为：

```
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
    list_entry_t *le = &free_list;//le 赋值为空闲链表头
    struct Page * p;
    while((le=list_next(le))!=&free_list){//找到释放的位置，即第一个大于base 的空闲块
        p=le2page(le,page_link);//块地址转换为页地址
        if(p>base){//找到后跳出循环
            break;
        }
    }
    //according the base addr of withdrawed blocks, search free list, find the correct position
    for(p=base;p<base+n;p++){
        p->flags=0;
        set_page_ref(p,0);
        ClearPageReserved(p);
        ClearPageProperty(p);
        //全部重置
        //reset the fields of pages, such as p->ref, p->flags (PageProperty)
        list_add_before(le,&(p->page_link));//将每一个空闲块对应的链表插入空闲链表中
    }
    //(from low to high addr), and insert the pages. (may use list_next, le2page, list_add_before)
    base->property=n;
    SetPageProperty(base);
    //如果是高位，则向高地址合并
    p=le2page(le,page_link) ;
    if(base+n==p){
        base->property+=p->property;
```

```
        p->property=0;
    }
    //如果是低位，则向低地址合并
    le=list_prev(&(base->page_link));
    p=le2page(le,page_link);
    if(le!=&free_list&&p==base-1){//满足条件，未分配则合并
        while(le!=&free_list){
            if(p->property!=0){
                p->property+=base->property;
                base->property=0;
                //不断更新 p 的 property 值，并将 base->property 置为 0
                break;
            }
            le=list_prev(le);
            p=le2page(le,page_link);
        }
    }
    //try to merge low addr or high addr blocks. Notice: should change
some pages's p->property correctly.
    nr_free+=n;//空闲页数量加 n
    return;
}
```

请回答如下问题：你的 first fit 算法是否有进一步的改进空间：

答：由于实验原代码的 first-fit 算法的基本思想是使用链表去进行增删改查，所以这四项操作的时间复杂度均为 O(n)。

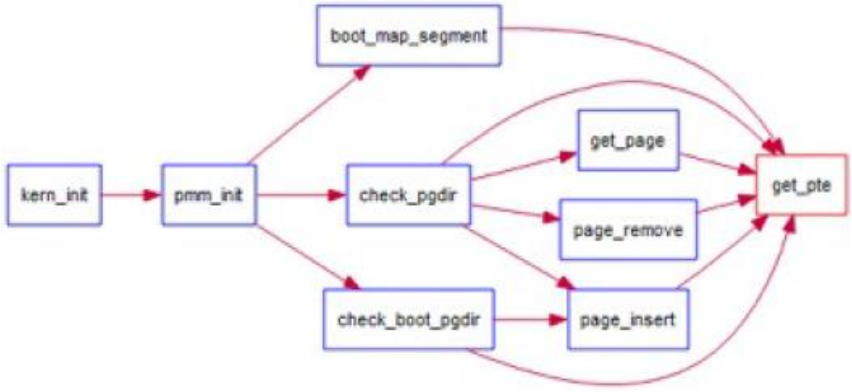
但我觉得可以将整体的数据结构修改为树形结构，这样就可以使得四项操作的时间复杂度降低至 O($log_2 n$)。但由于此种方法需要大幅度修改代码，考虑到时间成本，我并没有去亲手实现，但肯定是具有可行性的，只是需要更多的时间去调试运行。

同时，虽然书上有提到过 Best-fit（最佳适配），Next-fit（下次适配），但这两种方法在实际中的性能发挥并不如 first-fit（首次适配），所以不纳入改进方法的考虑范围内。

练习 2：实现寻找虚拟地址对应的页表项（需要编程）

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：



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？

答：写代码前先查看 mmu.h：

```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |     Index      |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \----------- PPN(la) -----------/
//
```

由上注释可知，32 位线性地址被分为 3 部分：

Page Directory：一级页表，高 10 位，可通过 PDX(la)获取；

Page Table：二级页表，中间 10 位，可通过 PTX(la)获取；

Offset within Page：页偏移量，低 12 位，可通过 PGOFF(la)获取。

所以易知每页大小为$2^{12} = 4096b = 4kb$.

根据注释即可写出 get_pte 的全部代码（设计实现过程也已放在代码注释中）：

```
//get_pte - 获取 pte 并为 la 返回此 pte 的内核虚拟地址
//        - 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
// parameter:
//  pgdir:  PDT 的内核虚拟基址
//  la:     线性地址需要映射
//  create: 逻辑值，以确定是否为 PT 分配页面
// return vaule: 该 pte 的内核虚拟地址
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
        /* LAB2 EXERCISE 2: YOUR CODE
     *
     * 如果您需要访问物理地址，请使用 KADDR（）
     * 请阅读 pmm.h 以获取有用的宏
     *
     * 也许您需要帮助注释，以下注释可以帮助您完成代码
     *
     * 一些有用的宏和定义，您可以在下面的实现中使用它们。
     * 宏或函数：
     *   PDX(la) = 返回虚拟地址 la 的页目录索引，即一级页表
     *   KADDR(pa) : 返回物理地址 pa 对应的虚拟地址
     *   set_page_ref(page,1) : 表示该页面被引用一次
     *   page2pa(page): : 获取 page 所管理的那一页的物理地址
     *   struct Page * alloc_page() : 分配页
     *   memset(void *s, char c, size_t n) : 将 s 指向的地址的前 n 个字节设
置为指定值 c。
     * DEFINEs:
     *   PTE_P           0x001                   // 对应地址的物理内存页
存在
     *   PTE_W           0x002                   // 对应地址的物理内存页
可写
     *   PTE_U           0x004                   // 对应地址的物理内存页
可读
     */
#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 allo
c 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
    pde_t *pdep=&pgdir[PDX(la)];// (1) find page directory entry
    //使用 PDX(la)，获取虚拟地址 la 的页目录索引（即一级页表位置）,再用 pgdir
来定位该 pte 的内核虚拟地址
    if (!(*pdep&PTE_P)){// (2) check if entry is not present
        //若该二级页表项不存在
        struct Page *page;
        //则需要根据 create 位的值来判断是否创建这个二级页表
        //create 为 0，不创建；反之则创建
        if(!create||(page=alloc_page())==NULL){// (3) check if creating
 is needed, then alloc page for page table
            return NULL;//若无需创建或分配页失败则返回 NULL
        }
        set_page_ref(page,1);// (4) set page reference
        //查找该页表时，引用次数+1
        uintptr_t pa=page2pa(page);// (5) get linear address of page
        //获取该页的物理地址
        memset(KADDR(pa),0,PGSIZE);// (6) clear page content using mems
et
        //物理地址转虚拟地址
        //由于该页虚拟地址未被映射，所以需要初始化
        *pdep = pa|PTE_P|PTE_U|PTE_W;// (7) set page directory entry's
permission
        //设置控制位（存在，可读，可写）
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];// (8) return pa
ge table entry
    //用 KADDR 返回二级页表的线性地址
}
```

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。

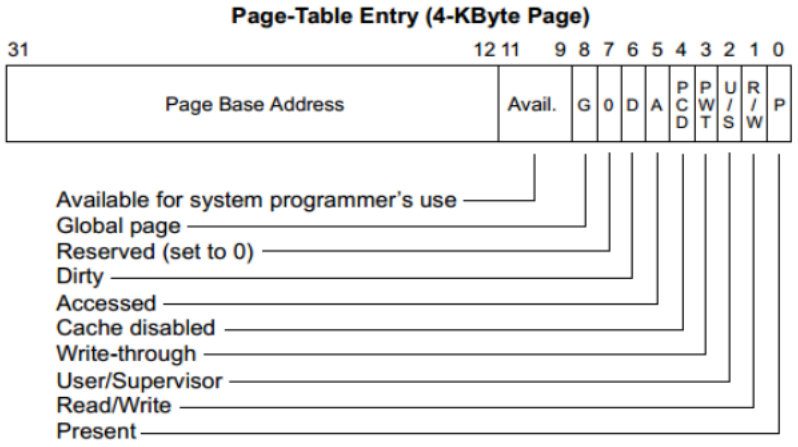
答：查看 mmu.h 可知页目录项和页表项的组成部分：

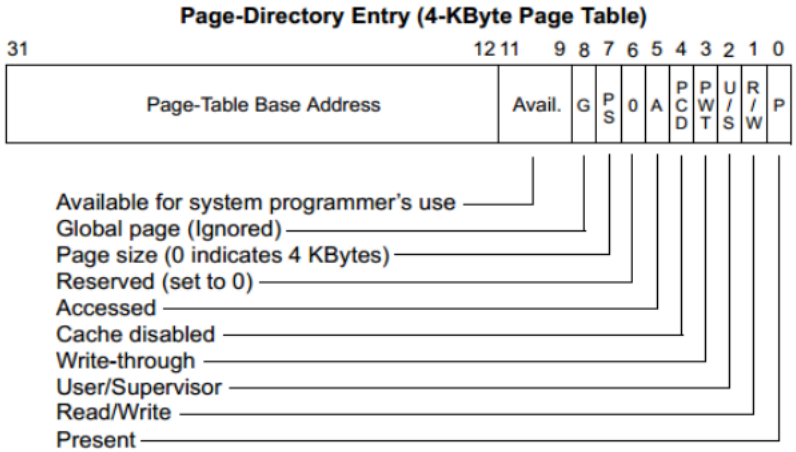
```
/* page table/directory entry flags */
#define PTE_P           0x001                   // Present
#define PTE_W           0x002                   // Writeable
#define PTE_U           0x004                   // User
#define PTE_PWT         0x008                   // Write-Through
#define PTE_PCD         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 softwa
re use
                                                // The PTE_AVAIL bits a
ren't used by the kernel or interpreted by the
                                                // hardware, so user pr
ocesses are allowed to set them arbitrarily.

#define PTE_USER        (PTE_U | PTE_W | PTE_P)
```

所以可总结出以下表格：

名称	含义
PTE_P	存在位
PTE_w	可写控制位
PTE_U	该页访问需要的特权级（用户或内核）
PTE_PWT	直写（Write-through）控制位
PTE_PCD	缓存控制位
PTE_A	访存控制位
PTE_D	脏位
PTE_PS	页大小
PTE_MBZ	必须为 0
PTE_AVAIL	设置内核或系统中断

同时，页目录项和页表项的格式如下所示：





对 ucore 的用处：

①PDE(页目录项)的高二十位表示该 PDE 对应的页表起始位置，即物理地址。

②PTE(页表项)的高二十位表示该 PTE 条目指向的物理页的物理地址。

③页目录项的作用：一级页表，存储了各二级页表的起始地址。页表是二级页表，存储了各个页的起始地址。

④页表项的作用：二级页表，存储各个页的起始地址。通过页目录项和页表项就可以将一个虚拟地址（线性地址）翻译为物理地址。

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？

答: ①将引发页访问异常的线性地址保存在 cr2 寄存器中；

②保护现场，将寄存器的值压入中断栈中；

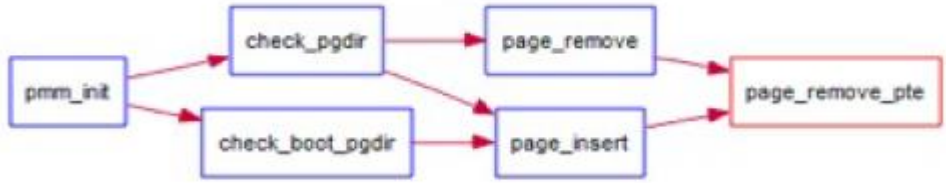
③设置 error_code（错误代码），将其也压入中断栈中；

④使得中断服务例程引发将外存的数据换到内存中来；

⑤进行上下文切换，退出中断，恢复到进入中断前的状态。

练习 3：释放某虚地址所在的页并取消对应二级页表项的映射（需要编程）

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：



答：根据注释即可写出 page_remove_pte 的全部代码（设计实现过程也已放在代码注释中）：

```
//page_remove_pte - 释放与线性地址 la 相关的 Page 结构
//                - 并清理（无效）与线性地址 la 相关的 pte
//note: PT 已更改，因此 TLB 需要设置为无效
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    /* LAB2 EXERCISE 3: YOUR CODE
     *
     * 请检查 ptep 是否有效，如果更新了映射，则必须手动更新 tlb
     *
     * 也许您需要帮助注释，以下注释可以帮助您完成代码
```

```
     *
     * 一些有用的宏和 define，您可以在下面的实现中使用它们。
     * 宏或函数:
     *   struct Page *page pte2page(*ptep): 从 ptep 的值获取相应的页面
     *   free_page : 释放页
     *   page_ref_dec(page) : 减少
page->ref. NOTICE: ff page->ref == 0 , 那么此页面应该被释放。
     *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : 使 TLB 条目无效，但
是仅在要编辑的页表是处理器当前正在使用的页表中。
     * DEFINEs:
     *   PTE_P           0x001                   // 页表/目录项标志位：存
在
     */
#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 re
ference reachs 0
                                  //(5) clear second page table entry
                                  //(6) flush tlb
    }
#endif
    if (*ptep&PTE_P){//(1) check if this page table entry is present
        //PTE_P 代表页存在，判断页表中该表项是否存在
        struct Page *page=pte2page(*ptep);//(2) find corresponding page
 to pte
        //获取该页
        if(page_ref_dec(page)==0){//(3) decrease page reference
        //判断是否只被引用了一次，若是 1 次的话，调用 page_ref_dec 减 1，值就为
0
            free_page(page);//(4) and free this page when page referenc
e reachs 0
            //若只被引用一次，则释放此页
            //因为为 0 的话，相当于不存在任何虚拟页指向该物理页
        }
        *ptep = 0;//(5) clear second page table entry
        //若被多次引用，则无需释放此页，只需释放对应的二级页表项
        //即设置二级页表项为 0，表示该映射关系无效
        tlb_invalidate(pgdir, la);//(6) flush tlb
        //刷新 TLB，保证 TLB 中的缓存不会有错误的映射关系
    }
}
```

在完成三个练习后，先执行 make grade，结果如下：

```
moocos-> make grade
Check PMM:                  (2.4s)
  -check pmm:                                   OK
  -check page table:                            OK
  -check ticks:                                 OK
Total Score: 50/50
```

再执行 make qemu：

```
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0105f96 (phys)
  edata  0xc0117a36 (phys)
  end    0xc0118968 (phys)
Kernel executable memory footprint: 99KB
ebp:0xc0116f38 eip:0xc01009d0 args:0x00010094 0x00000000 0xc0116f68 0xc01000bc
    kern/debug/kdebug.c:297: print_stackframe+21
ebp:0xc0116f48 eip:0xc0100cbf args:0x00000000 0x00000000 0x00000000 0xc0116fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0116f68 eip:0xc01000bc args:0x00000000 0xc0116f90 0xffff0000 0xc0116f94
    kern/init/init.c:49: grade_backtrace2+33
ebp:0xc0116f88 eip:0xc01000e5 args:0x00000000 0xffff0000 0xc0116fb4 0x00000029
    kern/init/init.c:54: grade_backtrace1+38
ebp:0xc0116fa8 eip:0xc0100103 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:59: grade_backtrace0+23
ebp:0xc0116fc8 eip:0xc0100128 args:0xc0105fbc 0xc0105fa0 0x00000f32 0x00000000
    kern/init/init.c:64: grade_backtrace+34
ebp:0xc0116ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    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, [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.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

若一开始不 make grade，直接 make qemu，则结果如下（结果基本与上图一致，只是在进入时钟中断后不会自动停止，也不会自动进入 kernel debug monitor）

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0105f70 (phys)
  edata  0xc0117a36 (phys)
  end    0xc0118968 (phys)
Kernel executable memory footprint: 99KB
ebp:0xc0116f38 eip:0xc01009d0 args:0x00010094 0x00000000 0xc0116f68 0xc01000bc
    kern/debug/kdebug.c:297: print_stackframe+21
ebp:0xc0116f48 eip:0xc0100cbf args:0x00000000 0x00000000 0x00000000 0xc0116fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0116f68 eip:0xc01000bc args:0x00000000 0xc0116f90 0xffff0000 0xc0116f94
    kern/init/init.c:49: grade_backtrace2+33
ebp:0xc0116f88 eip:0xc01000e5 args:0x00000000 0xffff0000 0xc0116fb4 0x00000029
    kern/init/init.c:54: grade_backtrace1+38
ebp:0xc0116fa8 eip:0xc0100103 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:59: grade_backtrace0+23
ebp:0xc0116fc8 eip:0xc0100128 args:0xc0105f9c 0xc0105f80 0x00000f32 0x00000000
    kern/init/init.c:64: grade_backtrace+34
ebp:0xc0116ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    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, [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
100 ticks
100 ticks
100 ticks
100 ticks
```

结果分析：通过上图，我们可以看到 ucore 在显示其 entry（入口地址）、etext（代码段截止处地址）、edata（数据段截止处地址）和 end（ucore 截止处地址）的值后，探测出计算机系统中的物理内存的布局（e820map 下的显示内容）。接下来 ucore 会以页为最小分配单位实现一个简单的内存分配管理，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最后显示 ucore 建立好的二级页表内容，并在分页模式下响应时钟中断。

将上面 4 张图与参考书上的结果对比，易知实验结果正确，代码无误。

3. 实验感想

本次实验工程量尚可，主要难度体现在对内存部分理论知识的掌握程上。代码实现部分只要能弄懂原理，再参考源码中本来就有的注释，一步一步来写的话肯定是可以完成的。

本次实验我个人觉得最难的还是练习 1 的 default_free_pages ( )，这一部分的代码相较于其他三个函数是最长也是最难理解的。我花了很久在这个函数上进行 debug，在参考了网上的代码并结合源码的指导注释之后才完成了这个函数（在函数中已经附上了具有自己理解的中文注释分析）。

这次实验主要讲的是物理内存的管理，因为在课上已经学习过相关理论知识，所以在实验中需要额外学习的知识并不多。主要需要查询的知识都集中在各种宏定义和函数的用法上，Page 和 list_entry_t 这两种数据结构以及 list.h 中的各种函数和宏定义都在此次实验中占了很大的比重。能否正确传入函数参数和理解宏的使用场景对实验结果来说非常重要。