操作系统课程设计实验报告

实验名称：清华大学操作系统实验 ucore lab2

姓名/学号：刘鑫/1120181208

一、 实验目的

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。在实验二中大家会了解并且自己动手完成一个简单的物理内存管理系统。

- 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

二、 实验内容

练习

为了实现 lab2 的目标，lab2 提供了 3 个基本练习和 2 个扩展练习，要求完成实验报告。

对实验报告的要求：

- 基于 markdown 格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 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
- 列出你认为本实验中重要的知识点，以及与对应的 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

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。

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

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：

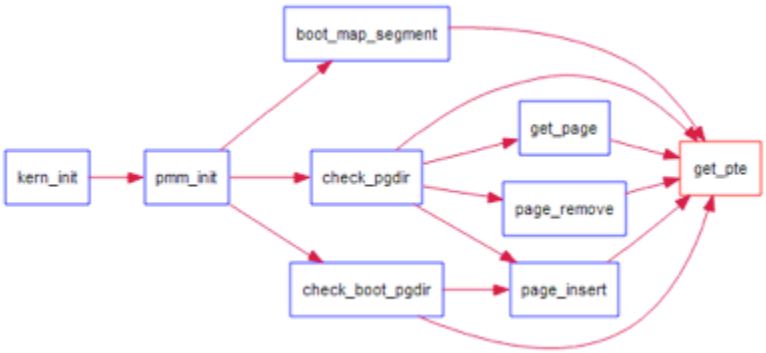

图 1 get_pte 函数的调用关系图

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？

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：

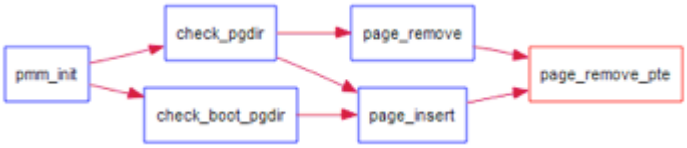

图 2 page_remove_pte 函数的调用关系图

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**

三、 实验环境

Windows：

Windows10

处理器 Inter core i5-8265u@1.60GHz

内存 8.00GB

系统类型 64 位操作系统，基于 x64 的处理器

Linux：

虚拟机软件：Oracle VM VitrualBox

虚拟机操作系统：ubuntu 14.04 x86-64

虚拟机内存：4GB

虚拟机硬盘容量:60GB

四、 程序设计与实现

练习一

实现 first-fit 连续物理内存分配算法

【思路】

首先 first-fit 分配算法按照地址从低到高查询空间的物理内存块，系统中的物理块采用双向链表的形式链接起来。在 mm/memlayout.h 中定义了结构 free_area_t 来实现双向链表

```
/* free_area_t - 包含一个用于记录空闲块（可分配的）的双向链表 */
typedef struct {
  list_entry_t free_list;       // 空闲块双向链表的头
  unsigned int nr_free;         // 空闲块的总数，严格来说是空闲页的总数
} free_area_t;
```

在 memlayout.h 中，还定义了另外一个关键的数据结构 Page 来完成空闲块的链接和信息存储

```
struct Page {
  int ref;                // 射到此物理页的虚拟页个数
  uint32_t flags;         // 此物理页的状态标记，比如是否被保留或者可分配等
  unsigned int property;      // 记录某连续内存空闲块的大小，在空闲块地址最小的一页起作用
  list_entry_t page_link;     // 用来链接多个空闲块，也是在空闲块地址最小的一页起作用
};
```

基于这个数据结构的管理物理页数组的起始地址就是全局变量 pages（定义在 mm/pmm.c 中，在 page_init()中使用）

操作系统的物理内存管理器就顺着双向循环链表进行搜索空闲区域内存，直到找到一个足够大的物理内存，如果这个内存和申请分配的内存刚好一样，就把这个内存块分配输出，否则就把物理空闲内存块一分为二，一部分与申请的内存大小一致，并把他分配出去，另一部分作为新的空闲物理内存块链接到空闲链表中

【实现】

1.default_init 函数

根据注释，可以直接使用原来的 default_init 函数，不需要做任何修改，函数如下图所示：

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

2. default_init_memmap 函数

根据函数的注释部分，可以直到，该函数作用是初始化某一个空闲块，参数为空闲块的起始地址和空闲块的页数，为了初始空闲块，需要初始化每一页，具体操作包括对页的状态位标志进行重置，选择地址空闲块地址最小的一页来存放空闲块中页的数目，然后使用 page_link 将该空闲块链接到空闲块链表，然后更新空闲块的总数。代码如下图所示：

```
static void
default_init_memmap(struct Page* base, size_t n) {
    assert(n > 0);  //判断n是否大于0
    struct Page* p = base;
    // 初始化n块物理页
    for (; p != base + n; p++) {
        assert(PageReserved(p)); // 检查此页是否为保留页
        p->flags = p->property = 0; // 标志位清0，将页标记为空闲
                                    // property表示空闲页面数，只有首页需要使用，因此置零
        set_page_ref(p, 0); // 清除引用此页的虚拟页的个数
    }
    nr_free += n; // 计算空闲页总数
    base->property = n; // 修改base的连续空页值为n
    SetPageProperty(base);
    list_add_before(&free_list, &(base->page_link));
}
```

3. default_alloc_pages 函数

由该函数的注释可知，该函数的主要功能是为进程分配，首先，寻找足够大的空闲块，，如果找到了，就重新设置标志位，然后冲空闲链表中删除此页，接着判断空闲块大小是否刚好等于所需内存大小，如果不是刚好等于，就分割页块，一部分刚好等于所需空闲块大小，并把这部分从空闲链表中去除。最后更新剩余空闲页个数，返回分配的页块地址。具体代码如下图所示：

```
static struct Page*
default_alloc_pages(size_t n) {
    assert(n > 0);
    // 如果页空间不足
    if (n > nr_free) {
        return NULL;
    }
    // 如果页空间足够分配,从头开始遍历保存空闲物理内存块的链表(按照物理地址的从小到大顺序)
    // 如果找到某一个连续内存块的大小不小于当前需要的连续内存块大小，则说明可以进行成功分配(选择第一个遇到的满足条件的空闲内存块来完成内存分配)
    struct Page* page = NULL;
    list_entry_t* le = &free_list;
    while ((le = list_next(le)) != &free_list) {   // 找到大于n的节点
        struct Page* p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    // 如果页空间足够分配
    if (page != NULL) {
        // 如果空闲空间中的空闲页面大于所需页面数，则将剩余页面分为两块
        // 第一块(物理地址较小的)用于分配，第二块加入空闲块链表
        if (page->property > n) {  // 如果大于n那么，截取前面的一部分
            struct Page* p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);   // 设置这一页的flags-PG_property为1，因为已经是头空闲块了
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));// 删除在空闲块链表中指向当前空闲块的项
        nr_free -= n;
        ClearPageProperty(page);   // 清除为0
    }
    return page;
}
```

4. default_free_pages 函数

该函数的实现的功能是释放以及使用完的页，把它们合并到 free_list 中。首先在 free_list 中寻找正确的插入位置，然后重置页状态标志，p->ref,p->flages,pageProperty 等等。最后检查 free_list 中在插入新的空闲页之后，有没有能够合并的页，若有则进行合并的操作。具体代码如下图所示：

```
static void
default_free_pages(struct Page* base, size_t n) {
    assert(n > 0);
    struct Page* p = base;
    for (; p != base + n; p++) {// 断言PG_reserved不是0，即这个是保留的，并且断言PG_property不是1，即已经分配了的
        assert(!PageReserved(p) && !PageProperty(p));
        // 将flags置为0并将ref置为0
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base); // 设置base的flags的PG_property为1，表示是头空闲页并且是free的
    list_entry_t* le = list_next(&free_list);  // 从next开始遍历双向链表
    while (le != &free_list) {
        p = le2page(le, page_link); // 获取这个节点的头空闲页Page结构
        le = list_next(le); // le指向下一个
        if (base + base->property == p) {  // 如果base基址加上空闲块的数目正好是p，说明base向上合并
            base->property += p->property;
            ClearPageProperty(p);  // 清除p这个页面的flags的PG_property为0，因为它不是头页面
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {  // 如果是向下合并的话，同理
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;  // 更新总空闲页数量
    le = &free_list;
    while ((le = list_next(le)) != &free_list) // 遍历双向链表，然后插入已经合并好的base
        if (base < le2page(le, page_link)) break;
    list_add_before(le, &base->page_link);
}
```

- **问题：你的 first fit 算法是否有进一步的改进空间**

  还存在进一步的改进空间，在上面使用的算法中，操作系统通过双向链表来查询空闲块，因此在查询空闲页的时间复杂度为 O(n),n 表示空闲块的数量，可以通过使用平衡二叉树来优化算法，查询的时间复杂度就会降低至 O(log N).

练习 2

【思路】

page directory enty 简称 pde_t,意为一级页表的表项，page table entry 简称为 pte_t,表示的是二级页表的表项，pgdir 表示的是页表的起始地址，unitptr_t 表示线性地址，因为段式管理只做直接映射，所以该地址也是逻辑地址。在引入进程后，每个进程都有一个自己的页表，但是可能存在没有对应的二级页表的情况，所以一开始并不需要分配二

级页表，而是等到需要的时候再添加二级页表。

【代码】

```
pte_t*
get_pte(pde_t* pgdir, uintptr_t la, bool create) {
    pde_t* pdep = &pgdir[PDX(la)];    // 获取到页目录表中给定线性地址对应到的页目录项
    if ((*pdep & PTE_P) == 0) {    // 判断页表是否存在
        if (!create) return NULL;  // 如果create为0不创建二级页表直接返回
        struct Page* page = alloc_page();  //创建二级页表
        if (page == NULL) return NULL;
        set_page_ref(page, 1);    // 设置物理页的引用为1
        uintptr_t pa = page2pa(page);  // 获取物理页的虚拟地址（此时已经启动了分页机制，内核地址空间）
        memset(KADDR(pa), 0, PGSIZE);  // 新创建的页表进行初始化
        *pdep = pa | PTE_U | PTE_W | PTE_P; // 设置页目录控制位
    }
    pte_t* pa = (pte_t*)PTE_ADDR(*pdep) + PTX(la); // 返回页的物理地址，我们找到的二级页表的入口，加上PTX(la)返回虚拟地址la的页表项索引就是中间10位
    return KADDR((uintptr_t)pa);   // KADDR输入物理地址进行转换，得到的就是页表项入口地址
}
```

【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？

（1）

Page Directory Entry 组成

- 第 0 位：PDE 的存在位
- 第 1 位：表示是否允许读写
- 第 2 位：表示该页访问所需要的特权级
- 第 3 位：设置是否使用 write through 缓存写策略
- 第 4 位：设置为 1，表示不对该页进行缓存
- 第 5 位：表示该页是否被使用过
- 第 6 位：为 0
- 第 7 位：设置 Page 大小，0 表示 4KB
- 第 8 位：可忽略
- 第 9-11 位：没有被 cpu 使用，可保留给操作系统使用
- 前 20 位：表示 4k 对齐的该 PDE 对应的页表的起始位置

Page Table Entry 组成

- 高 20 位与 PDE 相似，用于表示该 PTE 指向的物理页的物理地址
- 9-11 位保留给 OS 使用
- 7-8 位恒为 0
- 第 6 位表示该页是否为 dirty，即是否需要在 swap out 的时候写回外存
- 第 5 位表示是否被访问
- 3-4 位恒为 0
- 0-2 位分别表示存在位、是否允许读写、访问该页需要的特权级

PTE 和 TDE 都保留了一些位来供操作系统使用，ucore 利用这些保留位来完成内存管理的相关算法，有利于操作系统功能的拓展

(2)当发生页访问异常的时候，首先将发生错误的线性地址保存再 cr2 寄存器中，如果中断发生在内核态，则在中断栈中依次压入 EFLAGS，CS，EIP 和页访问异常码 error code，如果中断发生在用户态，还需要先压入 ss 和 esp，方便现场的恢复，然后切换到内核栈。根据中断描述符查表查询到对应的 page fault 的 ISR，然后跳转到对应的 ISR 处执行。

练习三

【思路】

通过判断该页被引用的次数来决定是否释放，如果只被引用了一次，则这个页就可以被释放

【代码】

```
static inline void
page_remove_pte(pde_t* pgdir, uintptr_t la, pte_t* ptep) {
    if (*ptep & PTE_P) {    // 判断这个页表中的页表项是否存在
        struct Page* page = pte2page(*ptep);   // 从这个页表项获取相对的页表
        if (page_ref_dec(page) == 0) { // 如果这个页表只被引用了一次那么直接释放这个页表
            free_page(page);
        }
        *ptep = 0; //    释放二级页表中这个页表项
        tlb_invalidate(pgdir, la); //    更新页表
    }
}
```

【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

对应关系，具体对应关系为：virtual address = line address = physical address + 0xc0000000

五、 实验收获与体会

本次实验主要内容是 80386 的内存管理，通过完成这次实验，我对课内学习的操作系统的关键的知识点有了更加深入的理解，从操作系统实际运行而非理论分析的角度理解了 OS 物理内存的管理方法，并参与设计了 first_fit 分配算法的功能设计与代码实现，整体上完成了一个简单的物理内存管理系统。