操作系统 Lab2 物理内存管理 实验报告

黄志鹏 PB16150288

- 实验目的
- 实验内容
- 基本练习
  - 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）
- 实验中涉及的知识点列举

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管理;
- 了解页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，对段页式内存管理机制有一个比较全面的了解;

基本练习

练习1:实现first-fit连续物理内存分配算法(需要编程)

1.1 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。

创建一个空的双向链表:

连续内存的维护由一个双向链表来实现，其结构如下图所示：

```
-------------------------------------------------------|
|   __    __    __    __    __    __    __    __     |
|--|  |->|  |->|  |->|  |->|  |->|  |->|  |->|  |<----
-->|__|<-|__|<-|__|<-|__|<-|__|<-|__|<-|__|<-|__|-----
|    \_free_area                                       |
-------------------------------------------------------
```

初始化时除了设置必要的位以外，只需将以base开始的连续n个page插入链表即可。之后可以通过与插入方向相反的指针顺序访问到这段连续的内存。
在参考答案中，是使用list_add_before来插入page，之后通过list_next来顺序访问。在本次实现中在插入过程中使用了list_add，所以之后的顺序访问就使用list_prev。两种实现方式在本质上并无区别。

分配内存时，从free_area->prev开始查找符合条件的内存块，找到即是返回的结果。如果内存块大于需求，还要进行分割。

回收内存时，仍然从头开始顺序查找，查找的目标是地址大于被释放内存块的地址的第一页。此处查找到结果，应先跳出查找目标的循环体，再做后续处理。（因为可能所有的内存空间都已经被占用，链表只剩free_area）被释放的内存要和前后相邻的空闲内存块合并。向后合并可根据其后一页是否与其相邻来判断。向前的合并可以一直向前便利直到找到一块空闲内存的头，然后与其合并即可。

对给定的`init`，`memmap`，`alloc_pages`和`free_pages`进行重写。

`default_init`：对操作系统要用的数据结构进行初始化，包括对free_list链表和nr_free空闲页个数置0。

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

`default_init_memmap`：对内存中空闲页的初始化，即将它们加入到空闲页的列表中。对于需要初始化的每一个页，设置标志位表明该页有效，property的值表示以该页开头有多少个连续的页，除了base外都为0，先统一设置成0，将页按顺序加到链表的末尾，再按需修改base的property和nr_free的值。

```
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));
}
```

`default_alloc_pages`：对从base开始的n个连续的页进行分配。从头开始遍历页表，一旦找到一个大于n个连续页的块，就开始分配。对于分配的项，设置标志表明它们已经被使用，在空闲页列表里删除当前页。如果当前空闲块的页数大于n，那么分配n个页后剩下的第一个页为新的块的形状，它的property比原来的小n。

```
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;
            SetPageProperty(p);
            list_add(list_prev(le), &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

`default_free_pages`：这个函数的作用是释放从base开始的n个连续页，并对空闲块做合并工作。按顺序寻找，找到第一个地址大于base的页的前面作为释放的块应插入的位置。对于要释放的每一个页，置引用个数为0，清除使用标志，并将它们重新加入的空闲链表当中。当前块是否能和后面的块进行合并，如果能，修改它们的property值，完成合并。

```
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);
    // Find insert location
    list_entry_t *next_entry = list_next(&free_list);
    while (next_entry != &free_list && le2page(next_entry, page_link) < base)
        next_entry = list_next(next_entry);
    // Merge block
    list_entry_t *prev_entry = list_prev(next_entry);
    list_entry_t *insert_entry = prev_entry;
    if (prev_entry != &free_list) {
        p = le2page(prev_entry, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            insert_entry = list_prev(prev_entry);
            list_del(prev_entry);
```

```
            }
        }
        if (next_entry != &free_list) {
            p = le2page(next_entry, page_link);
            if (base + base->property == p) {
                base->property += p->property;
                ClearPageProperty(p);
                list_del(next_entry);
            }
        }
        // Insert into free list
        nr_free += n;
        list_add(insert_entry, &(base->page_link));
    }
```

1.2 你的first fit算法是否有进一步的改进空间？

空闲页列表中存储的项是单页，而遍历的时候则是一页页遍历，通过property的值来判断块的大小，如果有一个块很大，那么需要遍历许多无用的页。可以考虑改进存储方式，将空闲页按连续块进行存储，而每一块中则只需要记录第一页的起始地址和块中页的个数，这样在遍历的时候可以节约空间。

练习2:实现寻找虚拟地址对应的页表项需要编程

实现过程：实现函数`get_pte(pde_t *pgdir, uintptr_t la, bool create)`

首先要查询一级页表，根据线性地址la在一级页表pgdir中寻找二级页表的起始地址。如果二级页表不存在，那么要根据参数create来分配二级页表的内存空间。

get_pte函数是通过PDT的基址pgdir和线性地址la来获取pte。PDX根据la获取其页目录的索引，根据此索引可以得到页目录项pde，由于可能对其进行修改，这里采用指向pde的指针pdep，而*pdep中保存的便是pde的真实内容。创建了pde后，需要返回的值是pte的指针，这里先将pde中的地址转化为程序可用的虚拟地址。将这个地址转化为pte数据类型的指针，然后根据la的值索引出对应的pte表项，最后通过&取得它的指针返回。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    if (!(pgdir[PDX(la)] & PTE_P)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
            return NULL;
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        pgdir[PDX(la)] = (pa & ~0xFFF) | PTE_P | PTE_W | PTE_U;
    }
    return (pte_t *)KADDR(PDE_ADDR(pgdir[PDX(la)])) + PTX(la);
}
```

2.1 请描述页目录项(Pag Director Entry)和页表(Page Table Entry)中每个组成部分的含义和以及对ucore而言的潜在用处。

PDE和PTE的组成（详见mmu.h文件）

```
#define PTE_P           0x001 // 当前项是否存在，用于判断缺页
#define PTE_W           0x002 // 当前项是否可写，标志权限
#define PTE_U           0x004 // 用户是否可获取，标志权限
#define PTE_PWT         0x008 // 写直达缓存机制,硬件使用Write Through
#define PTE_PCD         0x010 // 禁用缓存，硬件使用Cache-Disable
#define PTE_A           0x020 // Accessed
#define PTE_D           0x040 // 页是否被修改，硬件使用（dirty)
#define PTE_PS          0x080 // 页大小
#define PTE_MBZ         0x180 // 必须为0的位
#define PTE_AVAIL       0xE00 // 软件使用的位，可任意设置
```

因为页的映射是以物理页面为单位进行，所以页面对应的物理地址总是按照4096字节对齐的，物理地址低0-11位总是零，所以在页目录项和页表项中，低0-11位可以用于作为标志字段使用。

位	意义
0	表项有效标志（PTE_U）
1	可写标志（PTE_W）
2	用户访问权限标志（PTE_P）
3	写入标志（PTE_PWT）
4	禁用缓存标志（PTE_PCD）
5	访问标志（PTE_A）
6	脏页标志（PTE_D）
7	页大小标志（PTE_PS）
8	零位标志（PTE_MBZ）
11	软件可用标志（PTE_AVAIL）
12-31	页表起始物理地址/页起始物理地址

2.2 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？

出现了页访问异常的话，那么硬件将引发页访问异常的地址将被保存在cr2寄存器中，设置错误代码，触发page fault异常, 需要根据设置的IDT找到对应异常的处理例程的入口，然后跳转到该处理例程处理该异常。

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)

主要实现page_remove_pte函数：释放la地址所指向的页，并设置对应的pte的值.首先确保页存在,找到pte所在的页,把pte所在页的ref减一。如果该页已经没有ref，那么pte所在的页可以释放。释放pte值指向的页，再清空tlb即可。

```
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_P) {
        struct Page *page = pte2page(*ptep);
        if (page_ref_dec(page) == 0)
            free_page(page);
        *ptep = 0;
        tlb_invalidate(pgdir, la);
    }
}
```

3.1 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?

Page的全局变量pages对应的是每一个页的虚拟地址，而页表项和页目录项都指向一个页，它们保存的是页的物理地址，通过pte2page、pde2page可以将pte、pte中保存的页映射到page中的虚拟地址对应的页。

3.2 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

如果希望虚拟地址与物理地址相等，需要修改pages变量的地址，如果把它的虚拟地址映射到0处就实现了page的虚拟地址等于物理地址。 需要以下几个步骤

1. 在链接脚本中将内核的虚拟地址起始地址改为0x100000

```
/* /lab2/tools/kernel.ld */
SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0x100000;

...
```

2. 将虚拟内存起始地址改为0x0

```
/* /lab2/kern/mm/memlayout.h */

#define KERNBASE  0x00000000
```

3. 删除0-4M空间内存映射部分

```
/* /lab2/kern/mm/pmm.c

//disable the map of virtual_addr 0~4M
boot_pgdir[0] = 0;

//now the basic virtual memory map(see memalyout.h) is established.
//check the correctness of the basic virtual memory map.
check_boot_pgdir();

*/
```

在本次实验的测试过程中，仅完成练习二的get_pte()函数无法通过check_pgdir()函数的测试，需要完成练习三的任务才会通过全部测试。完成练习一的任务后即可通过check_alloc_page()的测试。

实验中涉及的知识点列举

列举本次实验中涉及到的知识点如下：

- 80386 CPU的段页式内存管理机制，以及进入页机制的方法；
- 对物理内存的探测的方法；
- 具体的连续物理内存分配算法，包括first-fit，best-fit等一系列策略；
- ucore中链表的实现方法；
- 在C语言中使用面向对象思想实现物理内存管理器；
- 链接地址、虚拟地址、线性地址、物理地址以及ELF二进制可执行文件中各个段的含义；

对应到的OS中的知识点如下：

- 内存页管理机制；
- 连续物理内存管理；

两者之间的对应关系为：

- 前者为后者提供了具体完成某一个平台上的操作系统对应的内存管理功能的底层支持；
- 同时在实验中设计到的其他一些知识点，比如面向对象思想、链表的使用等，方便了具体的操作系统的实现编码；