操作系统课程设计实验报告

实验名称： 物理内存管理

姓名/学号：宋尚儒/1120180717

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

本次实验包含三个部分。

- 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。

实验环境

- Windows10
- VMWare15.5
- Ubuntu-20.04
- ucore lab2
- qemu

程序设计与实现

练习0

> 填写已有实验

只需要将lab1中完成的代码移植到lab2框架中即可，主要涉及文件为

- `kern/debug/kdebug.c`
- `kern/trap/trap.c`

练习1

> **实现 first-fit 连续物理内存分配算法**

查看`default_pmm.c`中的各个函数之前，先查看需要用到的重要的结构和函数

```
//物理页框的属性结构
struct Page {
    int ref;                //该页框引用数
    uint32_t flags;         //物理页框属性数组，有两个标志位，第一位表示保留，第二位表示空闲
    unsigned int property;  //连续空闲页数(只在地址最低页有值)
```

```
    list_entry_t page_link; //双向链接结构
};

//双向链表结构
struct list_entry {
    struct list_entry *prev, *next; //前后节点
};

//物理内存管理器结构，包含多个管理指针，指向所需函数
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,                       //管理器初始化
    .init_memmap = default_init_memmap,         //空闲页初始化
    .alloc_pages = default_alloc_pages,         //分配指定数量的物理页
    .free_pages = default_free_pages,           //释放指定物理页
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};

//在双向链表元素prev和next间插入elm
static inline void
__list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t *next) {
    prev->next = next->prev = elm;
    elm->next = next;
    elm->prev = prev;
}

//同list_after
static inline void
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
}

//将新对象elm插入listelm前
static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}

//将新对象elm插入listelm后
static inline void
list_add_after(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm, listelm->next);
}

//从双向链表中删除该元素
static inline void
list_del(list_entry_t *listelm) {
    __list_del(listelm->prev, listelm->next);
}
```

连续空闲物理页框组成的空闲块会以双向链表的形式组织管理，具体而言，只有空闲块头部的页框对应的Page结构会指明该空闲块大小，其成员page_link构成双向链表结构。需要注意的是空闲块中非头部的页框是不会直接参与到以上组织管理结构中的。

接下来依次查看物理内存管理器结构属性指向的重要函数，分析物理内存管理、分配、回收过程

default_init

原函数为：

```
static void
default_init(void) {
    list_init(&free_list);     //初始化空闲块链表初始化
    nr_free = 0;               //设空闲页框总数为0
}
```

该操作与物理内存分配无直接关系，直接使用默认函数

default_init_memmap

作用为对从base开始的大小为n的未被占用的物理内存空间中的每一页框所对应的Page结构进行初始化

- 从base开始清零各Page结构的属性
- 设置base的连续页框数为n
- 由于采用ff策略，将base的双向链表成员加入空闲块链表尾部（具体操作为加入头部指针之前）

如此确保空闲块链表按地址从小到大排列

构造函数为：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    //循环初始化n块物理页
    for (; p != base + n; p ++) {
        //检查该页框是否为保留页
        assert(PageReserved(p));
        //将该页标志位和连续空闲页框数清零
        p->flags = p->property = 0;
        //设置该页框引用计数为0
        set_page_ref(p, 0);
    }
    //在第一个空闲页块中设置连续页数量为n
    base->property = n;
    //设置该页为保留页
    SetPageProperty(base);
    //计入空闲页数
    nr_free += n;
    //加入空闲页链表，由于采用ff策略，需要按地址从小到大排，需加到链表尾部，即头部之前
    list_add_before(&free_list, &(base->page_link));
}
```

default_alloc_pages

作用为分配指定页数的连续空闲空间，并返回空闲块头部页框对应的指针

- 在空闲块链表中找到第一个连续内存页数量大于n空闲块，其对应Page结构体的对象指针为page，指向该空闲块的第一个页框
- 从page开始划分n页，剩余的页框形成新的空闲块，加入空闲块链表中
- 从空闲块链表中删去page的空闲块链表指针
- 返回指针page

如此可以分配第一个符合要求的空闲块，并确保划分后新产生的空闲块不会影响链表中空闲块按地址从小到大排列的结构

构造函数为：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    //不可超过总空闲页框数
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    //遍历空闲块链表
    while ((le = list_next(le)) != &free_list) {
        //页链表结构指针转换为页结构指针
        struct Page *p = le2page(le, page_link);
        //找到连续内存页数量大于n的空闲块为止，保存为page
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    //分割原page指向的连续内存区域
    if (page != NULL) {
        if (page->property > n) {
            //要分配n页，分配后，剩余内存形成新的连续空闲块，指向分新划分出来的空闲块头部
            struct Page *p = page + n;
            //连续空闲块数量为原连续块数量减n
            p->property = page->property - n;
            //将新的连续块头部链表指针加入链表中，插到原页链表指针后
            list_add_after(&(page->page_link), &(p->page_link));
        }
        //从链表中删除该页链表
        list_del(&(page->page_link));
        //空闲块数量-n
        nr_free -= n;
        ClearPageProperty(page);
    }
    //返回page，即分配出的空闲内存区头部
    return page;
}
```

default_free_pages

作用为释放从base开始指定页数的内存空间，并分情况进行合并操作，需要考虑四种情况：

- 释放空间与其之前的空闲块相邻，与之后的不相邻
- 释放空间与其之前的空闲块不相邻，与之后的相邻
- 释放空间与其之前的空闲块不相邻，与之后的不相邻
- 释放空间与其之前的空闲块相邻，与之后的相邻

为使整体结构清晰以及操作的统一，采用以下形式合并

- 扫描空闲块链表，寻找与释放空间前后邻接的空闲块

  如果存在邻接空闲块，从链表中删去原空闲块，修改释放空间的属性

如果没有邻接空闲块，则第一步不会对链表产生影响

- 扫描空闲块链表，找到释放空间对应的空闲块在链表中应该放置的位置
- 将释放空间对应的空闲块加入链表

如此，可以正常回收空间并实现合并操作，不会影响链表地址中空闲块按地址从小到大排列的结构

构造函数为

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    //遍历内存页，清空数据
    for (; p != base + n; p ++) {
        //该页需要未保留、被使用
        assert(!PageReserved(p) && !PageProperty(p));
        //状态位清零
        p->flags = 0;
        //引用数清零
        set_page_ref(p, 0);
    }
    //设置起始内存页后连续空闲块的数量
    base->property = n;
    SetPageProperty(base);

    //采取不同策略合并连续空闲块
    //首先遍历空闲块链表
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        //p指向连续空闲页头部
        p = le2page(le, page_link);
        le = list_next(le);

        //如果释放的内存区域的尾部与一空闲区域头部邻接
        if (base + base->property == p) {
            base->property += p->property;   //加上该空闲内存区域
            ClearPageProperty(p);            //设为保留
            list_del(&(p->page_link));       //从链表中删除该空闲区域指针
        }
        //如果释放的内存区域的头部与一空闲区域尾部邻接
        else if (p + p->property == base) {
            p->property += base->property;   //更新该空闲内存页数量
            ClearPageProperty(base);         //设为保留
            base = p;                        //使base指向该空闲页头部
            list_del(&(p->page_link));       //从链表中删除该空闲区域指针
        }
    }
    //在空闲页链表中插入合成后的连续空闲页指针base
    //空闲块总数+n
    nr_free += n;
    //遍历空闲页链表
    le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);

        //如果合成的内存区域尾部小于等于空闲区域头部，保存该空闲区域对应的链表指针
        if (base + base->property <= p) {
```

```
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    //将base的链表指针插入寻找到的链表指针之前
    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

在本实验中的first fit算法可以做进一步改进，主要原因在于所有空闲块以双向链表的形式组织管理，在最坏的情况下需要找遍整个链表，假设链表长度为n，则时间复杂度达到O(N)

可以采用类似linux中伙伴系统的方式组织空闲块，具有相同大小的空闲块组织为一个链表，每次需要分配大小为n的空闲块时，直接从空闲块大小大于等于n的链表中查找空闲块，可以有效缩减查询所用时间。

练习2

> **实现寻找虚拟地址对应的页表项**

首先对内存地址的体系做简单概括，方便之后的理解。

内存地址分为三种：内核虚地址、线性地址、物理地址

- 内核虚地址：程序指令中使用的地址
- 线性地址：由三部分组成(页目录表地址，页目录项索引，页表索引)，可由内核虚地址通过段式管理地址映射得到
- 物理地址：实际访问的内存地址，可由线性地址通过页式管理地址映射得到

该练习需要构建`get_pte`函数，该函数三个参数的意义为：

- `pde_t *pgdir`：页目录表的内核虚拟基址
- `uintptr_t la`：需要映射的线性地址
- `bool create`：逻辑值，决定是否分配一个页

返回虚地址对应的页表项的内核虚地址，如果该页表项不存在，则还需要分配一个包含该项的页表

先看一下一些新的、重要的、需要直接调用的函数和宏定义

```
PDX(la)          //返回线性地址la对应的页目录索引
PTX(la)          //返回线性地址la对应的页表索引
KADDR(pa) :      //返回物理地址pa对应的内核虚拟地址
page2pa(page):   //得到Page结构管理的页的物理地址
PTE_P 0x001      //表示物理页存在
PTE_W 0x002      //表示物理页可写
PTE_U 0x004      //表示物理页可读
```

构造函数为

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    //获取页目录项
    pde_t *pdep = &pgdir[PDX(la)];
    //如果获取不成功，则需要创建新的页表
```

```
    if (!(*pdep & PTE_P)) {
        struct Page *page;
        //根据create判断是否创建，如果允许创建，则进行页分配，如果分配成功进行下一步
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        //设置新分配页的引用数为1
        set_page_ref(page, 1);
        //获取到该物理页的内核虚地址，并清空该页以初始化页表
        memset(KADDR(page2pa(page)), 0, PGSIZE);
        //对原先的页目录项对应的物理地址和操作权限进行设置
        *pdep = pa | PTE_U | PTE_W | PTE_P;
    }
    //返回线性地址la对应的页表项的内核虚地址
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
}
```

以下是两个问题的答案：

问题一

> 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。

页目录项PDE的组成为：

- 高20位：指向的页表基址的物理地址
- 9-11位：被CPU忽略，软件可用
- 8位：全局位，如果是全局的，则会在高速缓存中保存
- 7位：表示页大小，为0时表示4KB
- 6位：恒为0
- 5位：访问位
- 4位：表示能否对该页进行缓存，为1表示不可
- 3位：表示是否使用Write-Through缓存策略
- 2位：表示访问所需的用户特权级别
- 1位：表示是否允许读写，为0表示只读
- 0位：存在位，表示是否在内存中

页表项PTE的组成为：

- 高20位：指向的页的物理地址
- 9-11位：被CPU忽略，软件可用
- 8位：全局位，如果是全局的，则会在高速缓存中保存
- 7位：恒为0
- 6位：修改位，表示是否在该页写过数据，交换时是否修改外存
- 5位：访问位
- 4位：表示能否对该页进行缓存，为1表示不可
- 3位：表示是否使用Write-Through缓存策略
- 2位：表示访问所需的用户特权级别
- 1位：表示是否允许读写，为0表示只读
- 0位：存在位，表示是否在内存中

问题二

> 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?

当ucore执行过程中出现了页访问异常，硬件需要依次进行如下操作

- 保存发生异常时的寄存器现场，发生中断
- 产生页访问异常码，根据中断描述符表查询，并进入对应的异常处理程序，交由软件处理
- 处理完成后恢复上下文

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**

该练习需要重构`page_remove_pte`函数，该函数三个参数意义为

- `pde_t *pgdir`：页目录表的内核虚拟基址
- `uintptr_t la`：线性地址
- `pte_t *ptep`：指向页表项

该函数会在释放某物理内存页时，清理对应的Page结构，并将线性地址对应的页表项清除

先看一下一些新的、重要的、需要直接调用的函数和宏定义

```
struct Page *page pte2page(*ptep)           //从页表项获取对应的Page结构体
free_page                                   //释放Page
page_ref_dec(page)                          //使Page结构的引用数减一,返回减一后ref
tlb_invalidate(pde_t *pgdir, uintptr_t la) //当修改的页表是进程正在使用的页表时，使对应的TLB表项无效化
```

构造函数为

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    //如果传入的页表项存在
    if (*ptep & PTE_P) {
        //获取传入页表项对应物理页框的Page结构
        struct Page *page = pte2page(*ptep);
        //将该Page结构引用数减一，如果减到零则释放该Page结构以及对应的物理页
        if (!page_ref_dec(page)) {
            free_page(page);
        }
        //清零该页目录项
        *ptep = 0;
        //刷新TLB，无效化TLB中对应的页目录项
        tlb_invalidate(pgdir, la);
    }
}
```

问题一

> 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?

有对应关系，因为页表项中存放了对应物理页的物理地址，可通过该物理地址获取对应的Page数组的对应项。

可将页表项中的物理地址除以页的大小，然后乘以Page结构的大小以获取数组偏移量，Page数组基地址加偏移量即可得到对应Page项的地址

问题二

> 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?

物理地址和虚拟地址之间存在一定偏移(offset)，通常该偏移由操作系统决定，而在ucore中，该值定义在`kern/mm/memlayout.h`

```
#define KERNBASE            0xC0000000
```

这里KERNBASE即为虚拟地址空间的内核基址，即偏移，此处定义为0xC0000000

将该值修改为0即可完成题目要求

实验结果

运行`make qemu`（仅展示部分），出现页表信息

```
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: 07ee0000, [00100000, 07fdffff], type = 1.
  memory: 00020000, [07fe0000, 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
```

运行`make grade`，结果正常

```
ssr@ubuntu:~/my_ucore/mylab2/lab2$ make grade
Check PMM:                  (3.4s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check ticks:                               OK
Total Score: 50/50
```

参考答案分析

- 在练习1中，本实验代码与参考答案基本一致，实现细节上仅有list_add_after和list_add两个函数的不同，但实际上两个函数具有相同的功能，这点在前文中已做过说明
- 在练习2中，本实验代码仅在函数和宏调用方式上略有不同
- 再练习3中，代码实现较为简单，与参考答案没有区别

实验收获与体会

本次实验主要学习物理内存管理机制，了解了如何探测系统中的物理内存（虽然这点没有在练习中给出要求），学习物理内存管理分配方法，将之前学习到的分配算法进行实际应用，并对内存地址体系和映射转换方式有了较为全面的了解

有了学习lab1的经验，在学习lab2的过程中更为顺利了，并且lab2中很多知识在原理课中有说明，相对lab1友好了许多。但受限于考试复习，感觉学习的还不是非常透彻，部分函数和宏没有找到源码，只是大致了解其用途和调用关系，如果有机会希望能对源码做进一步的了解与掌握