lab2

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

练习 0：

需要更改的文件为 kdebug.c 和 trap.c，具体更改的代码如下：

kdebug.c:

```
uint32_t t_ebp = read_ebp();
    uint32_t t_eip = read_eip();
    int i,j;
    for(i = 0;i< STACKFRAME_DEPTH && t_ebp!=0;i++)
    {
        cprintf("ebp=%08x,eip=%08x,args:",t_ebp,t_eip);
        uint32_t *args = (uint32_t *)t_ebp +2;
        for(j = 0;j<4;j++)
        {
            cprintf("0x%08x",args[j]);
        }
        cprintf("\n");
        print_debuginfo(t_eip-1);
        t_eip = ((uint32_t *)t_ebp)[1];
        t_ebp = ((uint32_t *)t_ebp)[0];
    }
```

trap.c:

```
void trap(struct trapframe *tf) {
    ticks ++; //每一次时钟信号会使变量 ticks 加 1
        if (ticks==TICK_NUM) {//TICK_NUM 已经被预定义成了 100，每到 100 便调用 print_ticks()
函数打印
            ticks=0;
            print_ticks();
        } ticks ++; //每一次时钟信号会使变量 ticks 加 1
        if (ticks==TICK_NUM) {//TICK_NUM 已经被预定义成了 100，每到 100 便调用 print_ticks()
函数打印
            ticks=0;
            print_ticks();
        }
}
void idt_init(void) {
extern uintptr_t __vectors[];
    int i;
    //初始化 idt
    for(i=0;i<256;i++)
    {
        SETGATE(idt[i],0,GD_KTEXT,__vectors[i],DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK],0,GD_KTEXT,__vectors[T_SWITCH_TOK],DPL_USER);

SETGATE(idt[T_SWITCH_TOU],0,GD_KTEXT,__vectors[T_SWITCH_TOU],DPL_KERNEL);
    lidt(&idt_pd);
}
```

练习 1：实现 first-fit 连续物理内存分配算法（需要编程）

1-1、实现思路

物理内存页管理器顺着双向链表进行搜索空闲内存区域，直到找到一个足够大的空闲区域。如果空闲区域的大小和申请分配的大小正好一样，则把这个空闲区域分配出去，成功返回；否则将该空闲区分为两部分，一部分区域与申请分配的大小相等，把它分配出去，剩下的一部分区域形成新的空闲区。其释放内存的设计思路很简单，只需把这块区域重新放回双向链表中即可。在实现之前，我们要先了解几个需要用到数据结构。

1-2、物理页属性结构

```
struct Page {
    int ref;                    // page frame's reference counter
    uint32_t flags;             // array of flags that describe the status of the page
frame
    unsigned int property;      // the num of free block, used in first fit pm manager
    list_entry_t page_link;     // free list link
};
```

ref 表示这样页被页表的引用记数，就是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 加一。反之，若是解除，那就减一。

flags 表示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 的。如果设置为 1，表示这页是 free 的，可以被分配；如果设置为 0，表示这页已经被分配出去了，不能被再二次分配。

property 用来记录某连续内存空闲块的大小，这里需要注意的是用到此成员变量的这个 Page 一定是连续内存块的开始地址（第一页的地址）。

page_link 是便于把多个连续内存空闲块链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 来链接比它地址小和大的其他连续内存空闲块

1-3、双向链表

```
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

free_list 是一个 list_entry 结构的双向链表指针

nr_free 则记录当前空闲页的个数

1-4、代码实现

ucore 使用物理内存管理类 pmm_manager 来绑定一个函数，我们需要在该类中包含需要实现的函数的名字。

default_init 函数：

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

直接调用库函数 list_init 初始化掉 free_area_t（管理所有连续的空闲内存空间块的数据结构 free_area_t）的双向链表和空闲块数，不需要修改。

default_init_memmap 函数：

函数功能：这个函数实现的是一个根据现有的内存情况构建空闲块列表的初始状态的功能。用来初始化空闲页链表的,初始化每一个空闲页，然后计算空闲页的总数。

实现思路：传入物理页基地址，和物理页的个数(个数必须大于 0)，然后对每一块物理页进行设置：先判断是否为保留页，如果不是，则进行下一步。将标志位清 0，连续空页个数清 0，然后将标志位设置为 1，将引用此物理页的虚拟页的个数清 0。然后再加入空闲链表。最后计算空闲页的个数，修改物理基地址页的 property 的个数为 n。

```
static void default_init_memmap(struct Page *base, size_t n)
{
    assert(n > 0);    //判断个数是否大于 0
    struct Page *p = base;
    for (; p != base + n; p ++) {    //初始化 n 块物理页
        assert(PageReserved(p));    //检查本页是否为保留页
        p->flags = 0;                //标志位清 0
        SetPageProperty(p);              //标志位设置为 1
        p->property = 0;
        set_page_ref(p, 0);          //清空引用此页的虚拟页个数
        list_add_before(&free_list, &(p->page_link));//插入空闲页的链表里面
    }
    nr_free += n;   //计算空闲块总数
    base->property=n; //base 的连续空闲页值为 n
}
```

*Page default_alloc_pages 函数：

函数功能是寻找第一个能够满足需求的空闲块（First-fit）。先顺序查找，直到查找完或找到第一个满足条件的空闲块，当找到后，就将该块截断取出。因此可以明显的看出原代码的问题，当块被找到之后，只取出了第一页，而其他页并没有被取出，且没有被设为保留页，所以按照这个思路进行改动。

```
static struct Page *default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {   //如果所有的空闲页的加起来的大小都不够，那直接返回 NULL
        return NULL;
    }
    list_entry_t *le, *len;
    le = &free_list;      //从空闲块链表的头指针开始
    while((le=list_next(le)) != &free_list) {//依次往下寻找直到回到头指针处,即已经遍历一次
      struct Page *p = le2page(le, page_link);//将地址转换成页的结构
      if(p->property >= n){ //由于是 first-fit，则遇到的第一个大于 N 的块就选中即可
        int i;
        for(i=0;i<n;i++){//递归把选中的空闲块链表中的每一个页结构初始化
          len = list_next(le);
          struct Page *pp = le2page(le, page_link);
          SetPageReserved(pp);
          ClearPageProperty(pp);
          list_del(le);//从空闲页链表中删除这个双向链表指针
          le = len;
        }
        if(p->property>n){
          (le2page(le,page_link))->property = p->property - n;//如果选中的第一个连续的块大
于 n，只取其中的大小为 n 的块
        }
        ClearPageProperty(p);
        SetPageReserved(p);
        nr_free -= n;//当前空闲页的数目减 n
        return p;
      }
    }
    return NULL;//没有大于等于 n 的连续空闲页块，返回空
}
```

default_free_pages()函数：

这个函数的用处是将使用完的页进行回收。原代码的缺陷主要在三点，一是插入链表是并没有找到 base 相对应的位置，二是没有把页插入到空闲页表中，三是合并不合理。

```
static void default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base));
    list_entry_t *le = &free_list;
    struct Page *p;
    //查找 base 位置
    while((le=list_next(le)) != &free_list){
```

```
            p = le2page(le, page_link);
            if(p>base)
                break;
        }
        //将页插入到空闲页
        for(p = base; p<base+n; p++){
            p->flags = 0;
            set_page_ref(p, 0);
            ClearPageReserved(p);
            ClearPageProperty(p);
            list_add_before(le, &(p->page_link));
        }
        base->property = n;
        SetPageProperty(base);
        //高位地址合并
        p = le2page(le,page_link) ;
        if(base+n == p){
            base->property += p->property;
            p->property = 0;
        }
        //低位地址合并
        le = list_prev(&(base->page_link));
        p = le2page(le, page_link);
        if(p==base-1){
            while (le != &free_list) {
                if (p->property) {
                    p->property += base->property;
                    base->property = 0;
                    break;
                }
                le = list_prev(le);
                p = le2page(le, page_link);
            }
        }
        nr_free += n;
        return ;
}
```

1-5、是否还有进一步的改进空间

我认为还有提升空间。在 free 操作中，寻找需要 free 的 base 地址的时候，依靠的是遍历，通过改进算法，可以直接将 base 地址传入，无需遍历，直接找到位置开始操作，减少时间开销。

练习 2：实现寻找虚拟地址对应的页表项（需要编程）

2-1、编程实现

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    // typedef uintptr_t pde_t
    // PDX 左边 10 位(PDE）
    // PTX 中间 10 位(PTE)
    // KADDR - takes a physical address and returns the corresponding kernel virtual address
    // #define PTE_ADDR(pte)      ((uintptr_t)(pte) & ~0xFFF) address in page table or page directory entry
    // #define PDE_ADDR(pde)      PTE_ADDR(pde) address in page table or page directory entry
    // pdep: page dirtory
    pde_t *pdep = NULL;
    uintptr_t pde = PDX(la);
    pdep = &pgdir[pde];
    // 非 present 也就是不存在这样的 page（缺页），需要分配页
    if (!(*pdep & PTE_P)) {
        struct Page *p;
        // 如果不需要分配或者分配的页为 NULL
        if (!create || (p = alloc_page()) == NULL) {
            return NULL;
        }
        set_page_ref(p, 1);
        // page table 的索引值（PTE)
        uintptr_t pti = page2pa(p);

        // KADDR: takes a physical address and returns the corresponding kernel virtual address.
        memset(KADDR(pti), 0, sizeof(struct Page));

        // 相当于把物理地址给了 pdep
        // pdep: page directory entry point
        *pdep = pti | PTE_P | PTE_W | PTE_U;
    }

    // 先找到 pde address
    // address in page table or page directory entry
    // 0xFFF = 111111111111
    // ~0xFFF = 1111111111 1111111111 000000000000
    // #define PTE_ADDR(pte)      ((uintptr_t)(pte) & ~0xFFF)
    // #define PDE_ADDR(pde)      PTE_ADDR(pde)
```

```
    uintptr_t pa = PDE_ADDR(*pdep);
    // 再转换为虚拟地址（线性地址）
    // KADDR = pa >> 12 + 0xC0000000
    // 0xC0000000 = 11000000 00000000 00000000 00000000
    pte_t *pde_kva = KADDR(pa);

    // 需要映射的线性地址
    // 中间 10 位(PTE)
    uintptr_t need_to_map_ptx = PTX(la);
    return &pde_kva[need_to_map_ptx];
}
```

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？

会进行换页操作。首先 CPU 将产生页访问异常的线性地址放到 cr2 寄存器中，然后就是和普通的中断一样，保护现场，将寄存器的值压入栈中，设置错误代码 error_code，触发 Page Fault 异常，然后压入 error_code 中断服务例程，将外存的数据换到内存中来，最后退出中断，回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)

实现思路：主要就是先判断该页被引用的次数，如果只被引用了一次，那么直接释放掉这页， 否则就删掉二级页表的该表项，即该页的入口。我们先将物理页的引用数目减一，如果变为零，那么释放页面；然后将页目录项清零，刷新页表，即可取消页表映射。

```
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep)
{
 if (*ptep & PTE_P) {                //如果页表项存在
        struct Page *page = pte2page(*ptep);          //找到页表项
        page_ref_dec(page);
        if (page->ref == 0) {       //如果只有当前进程引用
            free_page(page);            //释放页
        }
        *ptep = 0;                      //设置页目录项为 0
        tlb_invalidate(pgdir, la);        //修改的页表是进程正在使用的页表，使其无效
    }
    }
```