一、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

二、实验内容

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？

实现过程及代码

首先合并lab1和lab2代码

运行后发现有错误，然后再打开文件对照原代码，

关键变量：

#define free\_list (free\_area.free\_list)//空闲块的链表，但是不指向具体页 #define nr\_free (free\_area.nr\_free)//空闲块的个数

关键函数：

list\_init(&free\_list);//初始化空闲块链表 SetPageProperty(base); ClearPageProperty(base); 关键宏：

le2page(le, page\_link);//由链表指针得到对应页的地址

1. Alloc pages:用firstfit算法寻找空闲块 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;         } }

(2) Alloc pages:删除空闲块，若有剩余则把剩余的部分插入空闲块链表

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));             //Excise 1 :My Code

            list\_add(page->page\_link.prev, &(p->page\_link));/////////应该要插在链表合适的位置   }

        nr\_free -= n;

        ClearPageProperty(page); }

(3) Free pages:删除指定块相邻的空闲块，合并成大空闲块 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));         }

if (p->property >= n) {             page = p;             break;         } }

(二)Alloc pages:删除空闲块，若有剩余则把剩余的部分插入空闲块链表

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));             //Excise 1 :My Code

            list\_add(page->page\_link.prev, &(p->page\_link));/////////应该要插在链表合适的位置   }

        nr\_free -= n;

        ClearPageProperty(page); }

(三)Free pages:删除指定块相邻的空闲块，合并成大空闲块 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));         }

练习2：实现寻找虚拟地址对应的页表项（需要编程）

关键的函数以及宏函数：

     \* PDX(la) = 返回虚拟地址la的页目录索引

      KADDR(pa) : 返回物理地址pa相关的内核虚拟地址      \*   set\_page\_ref(page,1) : 设置此页被引用了一次      \*   page2pa(page): 得到page管理的那一页的物理地址      \*   struct Page \* alloc\_page() : 分配一页出来

     \* memset(void \*s, char c, size\_t n) : 设置s指向地址的前面n个字节为字节‘c’.

    pde\_t \*pdep = &pgdir[PDX(la)];//得到页目录项     if (!(\*pdep & PTE\_P)) {//假如页目录项不存在         struct Page \*page;

        if (!create || (page = alloc\_page()) == NULL) {//假如不需要分配页或者分配页失败

return NULL;         }

        set\_page\_ref(page, 1);//设置该页被引用了一次         uintptr\_t pa = page2pa(page);//得到该页物理地址

        memset(KADDR(pa), 0, PGSIZE);//物理地址转虚拟地址，然后把该页初始化         \*pdep = pa | PTE\_U | PTE\_W | PTE\_P;//设置可读，可写，存在位     }

    return &((pte\_t \*)KADDR(PDE\_ADDR(\*pdep)))[PTX(la)];

    //KADDR(PDE\_ADDR(\*pdep))：这部分是由页目录项地址得到关联的页表物理地址，再转成虚拟地址

    //PTX(la)：返回虚拟地址la的页表项索引

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）

if (\*ptep & PTE\_P) {  //判断是否存在

        struct Page \*page = pte2page(\*ptep);

if (page\_ref\_dec(page) == 0) { //判断是否需要释放

 free\_page(page);

 }

  ptep = 0;  //初始化二级页表项

tlb\_invalidate(pgdir, la); //更新tlb

 }