姓名：张磊

学号：171491311

班级：硬件二班

实验题目：实验二

1. 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

二、实验内容

本次实验包含三个部分。首先了解如何发现系统中的物理内存；然后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；最后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。本实验里面实现的内存管理还是非常基本的，并没有涉及到对实际机器的优化，比如针对 cache 的优化等。如果大家有余力，尝试完成扩展练习

三、实验步骤

练习一

思路：

首先我们需要用一个数据结构来描述每个物理页（也称页帧），这里用了双向链表结构来表示每个页。链表头用 free\_area\_t结构来表示，包含了一个 list\_entry 结构的双向链表指针和记录当前空闲页的个数的无符号整型变量 nr\_free。

typedef struct {

list\_entry\_t free\_list;  // the list header

unsigned int nr\_free;  // # of free pages in this free list

} free\_area\_t;

接下来需要了解管理物理页的Page数据结构，这个数据结构也是实现连续物理内存分配算法的关键数据结构，可通过此数据结构来完成空闲块的链接和信息存储，而基于这个数据结构的管理物理页数组起始地址就是全局变量pages。

struct Page {

    int ref;                        // 映射此物理页的虚拟页个数

    uint32\_t flags;                 // 物理页属性（空或不空）

unsigned int property;          // 连续空页有多少(只在地址最低页有值，其余为0)

list\_entry\_t page\_link;  // 双向链接各个Page结构的page\_link双向链表(用于释放)

};

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，这是一种速度很快的算法，因为它尽可能少地搜索链表。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。

（二）、任务：

修改default\_init\_memmap()，default\_alloc\_pages()，default\_free\_pages()函数。

（三）、实现：

1、default\_init\_memmap()

static void

default\_init\_memmap(struct Page \*base, size\_t n) {

assert(n > 0);

struct Page \*p = base;

for (; p != base + n; p ++) {

//检查此页是否为保留页

assert(PageReserved(p));

//设置标志位

p->flags = p->property = 0;

SetPageProperty(p);

//清零此页的引用计数

set\_page\_ref(p, 0);

//将空闲页插入到链表

list\_add\_before(&free\_list, &(p->page\_link));

}

base->property = n;

//计算空闲页总数

nr\_free += n;

}

2、default\_alloc\_pages()

此函数是用于为进程分配空闲页。 其分配的步骤如下：

① 寻找足够大的空闲块 ，如果找到了，重新设置标志位

②从空闲链表中删除此页

③判断空闲块大小是否合适 ，如果不合适，分割页块 ，如果合适则不进行操作

④ 计算剩余空闲页个数

⑤ 返回分配的页块地址

static struct Page \*

default\_alloc\_pages(size\_t n) {

assert(n > 0);

if (n > nr\_free) {

return NULL;

}

list\_entry\_t \*len;

list\_entry\_t \*le = &free\_list;

//在空闲链表中寻找合适大小的页块

while ((le = list\_next(le)) != &free\_list) {

struct Page \*p = le2page(le, page\_link);

//找到了合适大小的页块

if (p->property >= n) {

int i;

for(i=0;i<n;i++){

len = list\_next(le);

//让pp指向分配的那一页

//le2page宏可以根据链表元素获得对应的Page指针p

struct Page \*pp = le2page(temp\_le, page\_link);

//设置每一页的标志位

SetPageReserved(pp);

ClearPageProperty(pp);

//清除free\_list中的链接

list\_del(le);

le = len;

}

if(p->property>n){

//分割的页需要重新设置空闲大小

(le2page(le,page\_link))->property = p->property - n;

}

//第一页重置标志位

ClearPageProperty(p);

SetPageReserved(p);

nr\_free -= n;

return p;

}

}

//否则分配失败

return NULL;

}

2、default\_free\_pages()

这个函数的作用是释放已经使用完的页，把他们合并到free\_list中。 具体步骤如下：

①在free\_list中查找合适的位置以供插入

②改变被释放页的标志位，以及头部的计数器

③尝试在free\_list中向高地址或低地址合并

static void

default\_free\_pages(struct Page \*base, size\_t n) {

assert(n > 0);

assert(PageReserved(base));

struct Page \*p = base;

//查找该插入的位置le

list\_entry\_t \*le = &free\_list;

while((le=list\_next(le)) != &free\_list){

p = le2page(le, page\_link);

if(p>base) break;

}

//向le之前插入n个页（空闲），并设置标志位

for (p = base;p<base+n;p++) {

list\_add\_before(le, &(p->page\_link));

p->flags = 0;

set\_page\_ref(p, 0);

ClearPageProperty(p);

SetPageProperty(p);

}

//将页块信息记录在头部

base->property = n;

//是否需要合并

//向高地址合并

p = le2page(le, page\_link);

if (base + n == p) {

base->property += p->property;

list\_del(&(p->page\_link));

}

//向低地址合并

le = list\_prev(&(base->page\_link));

p = le2page(le, page\_link);

//若低地址已分配则不需要合并

if(le!=&free\_list && 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；

}

练习二、查找虚拟地址对应页表项

（一）、思路：

pde\_t全称为 page directory entry，也就是一级页表的表项（注意：pgdir实际不是表 项，而是一级页表本身。实际上应该新定义一个类型pgd\_t来表示一级页表本身）。pte t全 称为 page table entry，表示二级页表的表项。uintptr t表示为线性地址，由于段式管理只做直接映射，所以它也是逻辑地址。

pgdir给出页表起始地址。通过查找这个页表，我们需要给出二级页表中对应项的地址。 虽然目前我们只有boot\_pgdir一个页表，但是引入进程的概念之后每个进程都会有自己的页 表。

有可能根本就没有对应的二级页表的情况，所以二级页表不必要一开始就分配，而是等到需要的时候再添加对应的二级页表。如果在查找二级页表项时，发现对应的二级页表不存在，则需要根据create参数的值来处理是否创建新的二级页表。如果create参数为0，则get\_pte返回NULL；如果create参数不为0，则get\_pte需要申请一个新的物理页（通过alloc\_page来实现，可在mm/pmm.h中找到它的定义），再在一级页表中添加页目录项指向表示二级页表的新物理页。注意，新申请的页必须全部设定为零，因为这个页所代表的虚拟地址都没有被映射。

当建立从一级页表到二级页表的映射时，需要注意设置控制位。这里应该设置同时设置 上PTE\_U、PTE\_W和PTE\_P（定义可在mm/mmu.h）。如果原来就有二级页表，或者新建立了页表，则只需返回对应项的地址即可。

（二）、实现：

pte\_t \*

get\_pte(pde\_t \*pgdir, uintptr\_t la, bool create) {

/\*

\* MACROs or Functions:

\* PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.

\* KADDR(pa) : takes a physical address and returns the corresponding kernel virtual address.

\* set\_page\_ref(page,1) : means the page be referenced by one time

\* page2pa(page): get the physical address of memory which this (struct Page \*) page manages

\* struct Page \* alloc\_page() : allocation a page

\*memset(void \*s, char c, size\_t n) : sets the first n bytes of the memory area pointed by s

\* to the specified value c.

\* DEFINEs:

\* PTE\_P 0x001 // page table/directory entry flags bit : Present

\* PTE\_W 0x002 // page table/directory entry flags bit : Writeable

\* PTE\_U 0x004 // page table/directory entry flags bit : User can access

\*/

//尝试获取页表，注：typedef uintptr\_t pte\_t;

pde\_t \*pdep = &pgdir[PDX(la)]; // (1) find page directory entry

//若获取不成功则执行下面的语句

if (!(\*pdep & PTE\_P)) {

//申请一页

struct Page \*page;

if(!creat || (page = all\_page())==NULL){

return NULL;

}

//引用次数需要加1

set\_page\_ref(page, 1);

//获取页的线性地址

uintptr\_t pa = page2pa(page);

memset(KADDR(pa), 0, PGSIZE);

//设置权限

\*pdep = pa | PTE\_U | PTE\_W | PTE\_P;

}

//返回页表地址

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

}

练习三、释放虚拟地址所在页,并取消对应二级页表映射

（一）、思路：

判断此页被引用的次数，如果仅仅被引用一次，则这个页也可以被释放。否则，只能释放页表入口。

（二）、实现：

static inline voidpage\_remove\_pte(pde\_t \*pgdir, uintptr\_t la, pte\_t \*ptep) {

/\* MACROs or Functions:

\*struct Page \*page pte2page(\*ptep): get the according page from the value of a ptep

\*free\_page : free a page

\*page\_ref\_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 , then this page should be free.

\*tlb\_invalidate(pde\_t \*pgdir, uintptr\_t la) : Invalidate a TLB entry, but only if the page tables being

\* edited are the ones currently in use by the processor.

\* DEFINEs:

\* PTE\_P 0x001 // page table/directory entry flags bit : Present

\*/

<span style="white-space:pre"> </span>//判断页表是否存在

if (\*ptep & PTE\_P){

<span style="white-space:pre"> </span>struct Page\* page = pte2page(\*ptep);

//判断此页是否被多次引用

<span style="white-space:pre"> </span>if (page\_ref\_dec(page)==0){

free\_page(page);

<span style="white-space:pre"> </span>}

<span style="white-space:pre"> </span>\*ptep = 0;

//释放pte

<span style="white-space:pre"> </span>tlb\_invalidate(pgdir, la);  <span style="font-family: 宋体; font-size: 10.5pt; letter-spacing: 0pt; line-height: 19.5pt; text-indent: 0pt;">}}</span>

