MIT 6.828 LAB2

薛犇 1500012752

Introduction

这个lab的主题是内存的管理，涉及到物理内存、虚拟内存、Kernel地址映射这三个部分。在这个lab中，我们会实现很多对

- Page
- Page Table Entry
- Page Directory Entry
- virtual address mapping

的操作。在完成这些操作后，我们会对物理、虚拟内存的映射关系有更加深入的了解。

Exercise 1

第一个练习要求我们补全一些物理页(Physical Page)的相关函数：

```
boot_alloc()
mem_init()
page_init()
page_alloc()
page_free()
```

这些函数是对物理页最基本的操作，我们一个个来分析：

function 1

```
boot_alloc()
```

虽然这个函数的目的是用来分配物理页，但是这是JOS在初始化虚拟内存的时候才会用到的函数，在后面分配物理页的时候，我们用的都是page_alloc()。

进入函数内部：它定义了一个静态变量

```
static char *nextfree;  // virtual address of next byte of free memory
```

维护的是下一块可以分配的物理内存地址。我们可以观察到这个变量的初始值是：

```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

这个end指向的是kernel bss段的末端，是第一块没有被分配任何kernel代码或者全局变量的空间。

接着，按照题意我们可以很快得补全代码：

```
if(n > 0){
    char * alloc_addr = nextfree;
    nextfree = ROUNDUP(nextfree + n, PGSIZE);
    return alloc_addr;
}
else if(n == 0){
    return nextfree;
}
```

function 2

```
mem_init()
```

现学现用，上一个函数boot_alloc是用来分配物理空间的，那么我们现在就要来分配一下。观察一下这个函数，它首先分配了一个物理页用做页目录，也就是一级页表。

```
// create initial page directory.
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

我们看到这里有一个把kern_pgdir也映射到自己内部的操作，这是一个比较哲理的问题，我们知道页目录本身也是一个页表(4KB)，那么，存放这个页表的页目录是谁呢？就好比一个人的父亲同时也会是另一个人的儿子，“父亲”的“父亲”是谁？在这里JOS(Windows操作系统也是这样)，就把这个身份赋予自己，自己做自己的“父亲”。

初始化页目录之后，我们就要初始化物理页的结构了，这里有一个数据结构

```
struct PageInfo *pages; // Physical page state array
```

存放了物理页的信息，我们查看这个结构体PageInfo：其实存放的就是下一个free page的地址，以及自己的引用次数。

那么这个数据结构是kernel自己维护的，所以也要给它分配一个空间，并按照要求清0。

```
pages = (struct PageInfo*)boot_alloc(npages*sizeof(struct PageInfo));
memset(pages, 0, npages*sizeof(struct PageInfo));
```

function 3

```
page_init()
```

要初始化保存物理页信息的数据结构page，这里主要涉及到哪些page是free的问题，也就是哪些page是不被一些其他数据结构占用的。

- 第一页是not free的，因为要保存IDT和BIOS的数据结构
- low memory(base memory, convention memory) , 也就是前640KB的剩下部分是free的

- 640KB到1024KB之间的384KB是not free的，要留给IO（VGA display, 16-bit devices, BIOS ROM等等）
- 从1024KB，也就是1MB开始的Extended Memory是free的？根据提示，说是kernel的空间不能碰，于是找到entry.S里有这么一段话：

```
    # We haven't set up virtual memory yet, so we're running from

    # the physical address the boot loader loaded the kernel at: 1MB

    # (plus a few bytes).  However, the C code is linked to run at

    # KERNBASE+1MB.  Hence, we set up a trivial page directory that

    # translates virtual addresses [KERNBASE, KERNBASE+4MB) to

    # physical addresses [0, 4MB).  This 4MB region will be

    # sufficient until we set up our real page table in mem_init
```

  于是就暴力地把前4MB都置成not free了(x， 暂时没有出错，两个check都是pass的。

代码如下：

```
size_t i;
for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    // 4MB, which is mentioned in ;entry.S
    if( i == 0 || (i >= IOPHYSMEM/PGSIZE && i < 1024)){
        continue;
    }
    page_free_list = &pages[i];
}
```

funtion 4

```
page_alloc()
```

根据题意，因为之前已经设置好了page_free_list，所以这里的page_alloc相当于是做了列表的修改操作，非常简单。

```
// if out of memory
if(page_free_list == 0){
    return NULL;
}
// if decide to alloc a page with zeros
if(alloc_flags && ALLOC_ZERO)
    memset(page2kva(page_free_list), '\0', PGSIZE);
// alloc free page and update page_free_list
struct PageInfo * alloc_addr = page_free_list;
page_free_list = page_free_list->pp_link;
alloc_addr->pp_link = NULL;

return alloc_addr;
```

function 5

```
page_free()
```

这个函数的目的是在本Page的引用次数pp_ref减到0的时候把这个page加入到free_page_list里，比较简单

```
if(pp->pp_ref!=0 || pp->pp_link!=NULL){ // sanity check
    panic("page free error!");
}
pp->pp_link = page_free_list;
page_free_list = pp;
```

Exercise 2

这个练习要求我们阅读80386手册，了解page translation的机制。这里我做了一些总结：

1. Page Translation, 只有当CR0的PG位置1的时候，才会启动。这是在操作系统做软件初始化的时候做的。如果操作系统想要实现多8086任务，页保护机制，或者页虚拟内存机制，必须把这位置1。
2. Page Frame：页帧，就是一段连续的4KB的物理内存空间。
3. Linear Address：

31-22	**21-12**	**11-0**
DIR	PAGE	OFFSET

   用DIR在Page Directory里找Dir Entry，
4. Page Tables：本身就是一个page，所以有4KB大小的空间，最多存1K个entry

   Page Tabe分两层：
   - 第一层是一个page directory，里面存1K个Page Table的地址。
   - 第二层是一个page table，里面存1K个Page 的映射地址
   - 每个Page有4K个空间，所以只要用一个page directory，就能覆盖整个80386的地址空间：

     1K * 1K * 4K = 4G

5. 当前page directory的物理地址存在CR3寄存器里。也叫做Page Directory Base Register(PDBR)
6. Page Table Entry的最后一位是present bit。如果是1,可以进行page translation。如果是0,而且系统尝试适用这个entry进行地址翻译的话，那么会触发一个page-not-present异常，这个异常能把需要的page载入物理内存。然后原来的指令会重新执行。
7. 倒数第三位是User/Supervisor bit。
    - U/S = 0：给Supervisor

    CPL决定current level，0,1,2是Supervisor level， 3是User level

    当S-mode下，所有的页都是可被寻址的，而U-mode只能访问属于它的那部分
8. 倒数第二位是Read/Write bit。
    - R/W =0 ： read only
    - R/W = 1：read/write access

    在S-mode下，所有的页都是可读可写的。而U-mode就要受制于R/W位了
9. LDT与GDT区别:
    - GDT: contains code and data segments, LDT desciptor, TSS, descriptor
    - each task has its own LDT:LDT用来记录每个程序怎么分配自己被“许诺（欺骗）”的4G空间，比如代码段在哪里，数据段在哪里。
    - linux下特有的平坦模式：flat address。只有两种段，代码段和数据段，在4G空间重合，然后所有的保护措施都在分页模式下做。
10. protect mode:
    - 16-bit（80286）：segmentation, 把real mode下的selector（直接左移4位）改成去一个Descriptor Table里寻找段的基址（牵扯到GDT， LDT等等）
    - 32-bit（80386）：page机制
11. CPL, DPL, RPL:
    - Current Privilege Level：当前CPU执行程序的特权级，CS与SS上
    - Descriptor：表示某一个段或者某一个门的特权级，一般存储在GDT，LDT 上
    - Request：一个访问请求的特权级，在段描述子的01位。为了防止CPL低的程序间接调用可以访问高DPL的程序（如fread等），所以在程序调用的时候，会把CPL抄过来作为RPL，连带着一起看。

Exercise 4

这个部分需要我们完成以下函数：

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

function 1

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
```

可以从传入的参数类型和返回值类型看到，这个函数的作用是给定一个page directory，和虚拟地址va，返回这个va所在的page table entry（PTE）。只要理清楚了两层页表之间的关系，这个函数并不难实现。值得一提的是，在inc/mmu.h里定义的很多宏对这个exercise的函数编写真的很有用处……我一开始没有仔细看，吃了很多亏。

```
// page number field of address
#define PGNUM(la)    (((uintptr_t) (la)) >> PTXSHIFT)

// page directory index
#define PDX(la)      ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(la)      ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)

// offset in page
#define PGOFF(la)    (((uintptr_t) (la)) & 0xFFF)

// construct linear address from indexes and offset
#define PGADDR(d, t, o)  ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

根据要求，我们先要找到pde，pde其实就是Page directory里的一个项，跟取数组一样就能取出来，那至于是哪个项，就是va里的22～31位，根据PDX这个宏就能得到：

```
uintptr_t vaddr = (uintptr_t)va;
pde_t * pde = &pgdir[PDX(vaddr)];
```

接着，如果这个pde不存在的话，那么要根据传入的参数create，决定要不要新建一个pde，新建的过程包含两步：

- 给pde实际分配一个物理页
- 把在pde中相应的位置插入这个va相对应的pte

```
if(!(*pde & PTE_P)){
    if(create == 0)
        return NULL;
    else{
        struct PageInfo* page_addr = page_alloc(ALLOC_ZERO); // alloc a page for pde
        if(page_addr == NULL){
            return NULL;
        }
        page_addr->pp_ref ++;
        *(pde) = page2pa(page_addr);
        // insert pte into pde
        *pde |= PTE_W | PTE_P | PTE_U; // be more tolerant ! ! !
        p = (pte_t*)KADDR(PTE_ADDR(*pde));
        return p + PTX(vaddr);
    }
}
```

这里实现的时候还有一个小插曲，关于给pde权限的设置，发现要尽量给多的权限，比如PTE_U。我是在做完lab3之后才开始写Lab2的report的，所以这里遇到一个坑……就是如果不设PTE_U，lab2的test都是可以过的，但是到涉及到了user/kernel的Lab3，就会出现蜜汁bug，找了好久才发现这里没有设置PTE_U，找了好久！！！具体就是，一个va明明我打印出它的权限的时候发现它是PTE_U的，但是就是会page fault，大概花了一天的时间才搞明白原来这只是pte的权限，pte上面的pde还不是PTE_U的，所以会page fault。吐血。。。

function 2

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
```

这个函数是用来把va开始size长度的虚拟地址映射到pa开始size长度的物理地址，利用之前实现的pgdir_walk函数，就能获得va所在的pte，把这个pte里面的值设成期望的pa，然后再对这个pte做一些权限的设置就行了。

```
int map_num = ROUNDUP(size, PGSIZE) / PGSIZE;
int pp = 0;
for(;pp<map_num;pp++){
    uintptr_t tmp_va = va + (pp << PGSHIFT);
    pte_t * ptb = pgdir_walk(pgdir, (void*)tmp_va, 1);
    *ptb = pa + (pp << PGSHIFT);
    *ptb |= perm | PTE_P;
}
```

这个函数非常有用，会在之后很多地方用到。

function 3

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
```

查询va所在的物理页page。如果pte_store不为空，那么就把pte的信息存进这里。个人感觉这个函数是让大家熟悉pa2page这个小函数的用法：

```
static inline struct PageInfo*
pa2page(physaddr_t pa)
{
    if (PGNUM(pa) >= npages)
        panic("pa2page called with invalid pa" );
    return &pages[PGNUM(pa)];
}
```

能够非常方便的将pa和它在Page列表里的信息对应起来。

至于page_lookup，就是先用pgdir_walk把pte找到，然后取个地址就把pa找到，接着再用pa2page就行了

```
pte_t * ptb = pgdir_walk(pgdir, va, 0);
if(ptb == NULL)
    return NULL;

uint32_t perm = *(ptb) & 0xfff;
```

```
physaddr_t pa = PTE_ADDR(*ptb);
// no page mapped at va
if((perm & PTE_P) == 0){
    return NULL;
}

if(pte_store != 0){
    *(pte_store) = ptb;
}

return pa2page(pa);
```

function 4

```
void
page_remove(pde_t *pgdir, void *va)
```

取消va处的地址映射。

```
pte_t * ptb = NULL;
struct PageInfo * pp = page_lookup(pgdir, va, &ptb);
page_decref(pp);
if(ptb != NULL){ // sanity check
    *(ptb) = 0;
}
tlb_invalidate(pgdir, va);
```

function 5

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
```

```
pte_t * ptb = pgdir_walk(pgdir, va, 1);
if(ptb == NULL){
    return -E_NO_MEM;
}
uintptr_t vaddr = (uintptr_t)va;
physaddr_t pa = PTE_ADDR(*ptb);
if((*ptb & PTE_P) == 1){
    if(pa != page2pa(pp)){
        page_remove(pgdir, va);
    }
    else{
        pp->pp_ref--;
    }
}
*ptb = page2pa(pp);
*ptb |= (perm | PTE_P);
pp->pp_ref++;

return 0;
```

这个函数是要把一个page映射到va上，这个函数最trick的地方在于：假如va本身有映射到某个page上，那么要把这个page给remove掉，但是，如果va已经映射到我想要映射的page上了，那么重复remove再insert会导致一些奇怪的错误，对于这种特殊情况，并不需要实际调用page_remove，特判一下，再对pp_ref操作一下，让它保持不变就可以了。

Exercise 5

利用boot_map_region，完成要求的映射：

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);

boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W
|PTE_P);

boot_map_region(kern_pgdir, KERNBASE, 1<<28, 0, PTE_W | PTE_P);
```

Chanllenge

showmappings

```
int debuginfo_VMmapping(uintptr_t va, struct VMmappinginfo *info)
{
    pte_t * ptb = pgdir_walk(kern_pgdir, (void*)va, 0);
    info->va = va;
    if(ptb == NULL){
        info->pa = 0;
        info->perm = 0;
        return 0;
    }

    uint32_t perm = *(ptb) & 0xfff;
    physaddr_t pa = PTE_ADDR(*ptb);
    info->pa = pa;
    info->perm = perm;
    return 0;
}
```

在kdebug.c里加入了如下函数，利用这个函数可以在知道[va, va+len]这个范围内的虚拟地址的权限情况。

```
K> showmappings 0xf0001000 0xf0002000
Trying to show mappings  in vm range [0xf0001000, 0xf0002000]
va:f0001000 pa:1000 perm: PTE_W PTE_P
va:f0002000 pa:2000 perm: PTE_W PTE_P
```

setperm

```
int debug_set_VMperm(uintptr_t va, int perm)
{
    pte_t * ptb = pgdir_walk(kern_pgdir, (void*)va, 0);
    if(ptb == NULL)
        return -1;
    *ptb |= perm;
    return 0;
}
```

利用这个函数可以在命令行修改va的地址的权限。

```
k>setperm 0xf0001000 w //  把0xf0001000置为可写
k>setperm 0xf0001000 u //  把0xf0001000置为用户态
k>setperm 0xf0001000 !w //  把0xf0001000置为不可写
k>setperm 0xf0001000 !u //  把0xf0001000置为核心态
```

dumpVM

打印[va, va+len]这个范围内的内容。

```
K>dumpVM -p 0x1000 0x2000 // 打印物理地址0x1000~0x2000 范围内的内容
K>dumpVM -v 0xf0001000 0xf0003000 //  打印虚拟地址0xf0001000~0xf0003000 范围内的内容
```

以上三个命令都可以使用。

Question

```
1. Assuming that the following JOS kernel code is correct, what type should variable x
have, uintptr_t or physaddr_t?

        mystery_t x;
        char* value = return_a_pointer();
        *value = 10;
        x = (mystery_t) value;
```

我认为是uintptr_t，因为在c代码里定义的指针都是virtual address。

```
2. What entries (rows) in the page directory have been filled in at this point? What
addresses do they map and where do they point? In other words, fill out this table as
much as possible:
Entry   Base Virtual Address    Points to (logically):
1023    ?       Page table for top 4MB of phys memory
1022    ?       ?
.       ?       ?
.       ?       ?
.       ?       ?
2       0x00800000      ?
1       0x00400000      ?
0       0x00000000      [see next question]
```

```
 UPAGES             : [0xef000000,0xef400000)    ->[pages    -0xf0000000,pages
-0xefc00000) cprintf 得到[0x0011a000,0x0015a000)
 KSTACKTOP-KSTKSIZE : [0xefbf8000,0xefc00000)    ->[bootstack-0xf0000000,bootstack-
0xefff8000) cprintf 得到[0x0010e000,0x00116000)
 KERNBASE           : [0xf0000000,0x100000000)   ->[0,0x10000000)
```

取虚拟地址前十位分别是508`0111 1111 00b`,510`0111 1111 10b`,960~1023`0x1111 0000 00~0x1111 1111 11`

还有作者之前手工映射的UVPT`[0xef400000,0xef800000)`->`[kern_pgdir-0xf0000000,kern_pgdir-0xefc00000)`

```
3.We have placed the kernel and user environment in the same address space. Why will
user programs not be able to read or write the kernel's memory? What specific
mechanisms protect the kernel memory?
```

因为user程序崩坏的话最多只是程序终止，但如果Kernel程序崩坏，整个计算机就瘫痪了，所以必须保证user和kernel之间的隔离。PTE_U的设置能够实现这种保护。

```
4. What is the maximum amount of physical memory that this operating system can
support? Why?
```

2G。因为UPAGES的最大尺寸是4MB，sizeof(struct PageInfo) = 8 Byte。所以最多支持512K个page，一个Page 4K，那么总共最多能支持2G。

```
5. How much space overhead is there for managing memory, if we actually had the maximum
amount of physical memory? How is this overhead broken down?
```

问我们需要多少空间来维护内存：

- PD 4K
- PT 2MB
- pages 4MB

如果使用大页，可以减少一些开销。

```
6. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately
after we turn on paging, EIP is still a low number (a little over 1MB). At what point
do we transition to running at an EIP above KERNBASE? What makes it possible for us to
continue executing at a low EIP between when we enable paging and when we begin running
at an EIP above KERNBASE? Why is this transition necessary?
```

在jmp指令结束后。

有必要，因为那之后用的都是虚拟地址，如果不保持这种映射，会因为找不到物理地址而出错。