实验 4 内存管理/虚拟内存

班级： 谢志鹏 姓名： 王傲 学号： 15300240004

请于 2017 年 11 月 28 日实验课后当天内，上传到 ftp://10.141.251.211。
（用户名：stu，密码：os）

实验 1.
阅读材料，熟悉 kern/pmap.ch 中的代码。mem_init()函数是为对系统初始化的函数，请熟悉 mem_init()然后分别填写以下代码

1：boot_alloc()函数缺少的代码

```
    if (n != 0)
    {//if n>0, allocates enough pages of contiguous physical memory to hold
'n' bytes
        char *next = nextfree;

        nextfree = ROUNDUP((char *) (nextfree+n), PGSIZE);
        //allocate a chunk large enough to hold 'n' bytes, then update
nextfree
        //keep aligned to a multiple of PGSIZE
        return next;
        //returns the address of the next free page
    }
    else
        return nextfree;
        //directly returns the address of the next free page
```

boot_alloc()的作用是在 JOS 初始化虚拟内存时，分配物理内存。当 n 大于 0 时，分配足以容纳下 n 个字节的连续的页（4K 对齐），当 n=0 时直接返回下一个可用的页。这里使用了一个指针 nextfree 指向下一个可用的页（初始时默认为 0），当 n 不为 0 时，计算出对齐后需要的内存大小，记下当前空闲块（即要被分配的块）的头部地址，同时更新 nextfree,表示这段内存已经被分配，最终返回被分配的块的首部地址。

特别说明，boot_alloc()函数用到了 end，由链接器分配，永远指向内核的 BSS 段的末尾，即链接器没有分配给任何内核代码或者全局变量的虚拟地址的部分的第一个虚拟地址。nextfree 利用 end 来进行更新。

2：mem_init()中缺少的代码（功能是调用 boot_alloc()函数分配一块内存，用来存放一个 struct PageInfo 的数组）

```
    pages = (struct PageInfo *) boot_alloc(sizeof(struct PageInfo) *
npages);
    //allocate an array of npages 'struct PageInfo's and store it in 'pages'
    memset(pages, 0, sizeof(struct PageInfo) * npages);
    //use memset to initialize all fields of each struct PageInfo to 0
```

这段代码比较简单，调用 boot_alloc()函数分配一段长为 npages 个 struct PageInfo 的物理内存，并将内存填充为 0。

实验 2.

参考相关资料，根据注释补全 mem_init()函数中有关页初始化，页分配和页释放的代码（page_init()，page_alloc(),page_free()函数）

1：page_init()

```
    size_t i;
    uint32_t pa;
    page_free_list = NULL;
    for (i = 0; i < npages; i++)
    {
        if(i == 0)
        {
        //mark physical page 0 as in use
        //this way we preserve the real-mode IDT and BIOS structures in case
we ever need them
            pages[0].pp_ref = 1;
             //set the reference as 1 time
            pages[0].pp_link = NULL;
            continue;
        }
        else if(i < npages_basemem)
        {
        // The rest of base memory, [PGSIZE, npages_basemem * PGSIZE) is
free
            pages[i].pp_ref = 0;
             //set the reference as 0 time
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
             //insert into the list
        }
```

```
        else if(i <= (EXTPHYSMEM/PGSIZE) || i < (((uint32_t)boot_alloc(0) -
KERNBASE) >> PGSHIFT))
        {
        // At IOPHYSMEM (640K) there is a 384K hole for I/O.  From the
kernel,
        // IOPHYSMEM can be addressed at KERNBASE + IOPHYSMEM.  The hole ends
        // at physical address EXTPHYSMEM.
        //#define IOPHYSMEM    0x0A0000
        //#define EXTPHYSMEM    0x100000
        //the IO hole [IOPHYSMEM, EXTPHYSMEM] must never be allocated

            pages[i].pp_ref++;
            pages[i].pp_link = NULL;
             //don't insert into the page_free_list
        }
        else
        {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }

        pa = page2pa(&pages[i]);
        //map a struct PageInfo * to the corresponding physical address

        if((pa == 0 || (pa >= IOPHYSMEM && pa <= ((uint32_t)boot_alloc(0) -
KERNBASE) >> PGSHIFT )) && (pages[i].pp_ref == 0))
        {
            cprintf("page error: i %d\n",i);
        }
    }
```

page_init()函数用来初始化页式地址管理和空闲页表队列，即要初始化 pages 数组和 page_free_list。我们可以看到这个 page_free_list 指向了所有的 Page 结构，建立其每个物理页面对应的实际链表节点，我们要把那些被操作系统占用或是系统预留空间从链表里去除掉。这里只需要剔除两块地址，第一块是 0 地址开始的第一个页面，第二块就是 I/O hole 开始的向上的一组连续的页面。这里 page0 要保存 IDT 和 BIOS 的相关结构，不能被分配，所以我们只是设置 page0 的 pp_ref 为 1，同时设置 pp_link 为 NULL；对于 I/O hole，[IOPHYSMEM, EXTPHYSMEM]段对应的物理地址要被保留，不能被分配；其余部分可以正常分配，pp_ref 为 0，

设置 pp_link 加入 page_free_link。使用 page2pa 将一个物理地址同一个 struct PageInfo 进行映射。

2：page_alloc()

```
    if (page_free_list)
    {//if there is free page in the page_free_list:
        struct PageInfo *ret = page_free_list;
        //get the first free page
        page_free_list = page_free_list->pp_link;
        //update the start of the page_free_list
        if (alloc_flags & ALLOC_ZERO)
            memset(page2kva(ret), 0, PGSIZE);
        //if (alloc_flags & ALLOC_ZERO), fills the entire returned physical
page with '\0' bytes
        ret->pp_link = NULL;
        //set the pp_link field of the allocated page to NULL
        //so page_free can check for double-free bugs
        return ret;
    }
    return NULL;
```

分配一个物理页，这里的做法是去掉 page_free_list 上的首个页并用于分配。相应的调整 page_free_list 获得要分配的页，如果（alloc_flags & ALLOC_ZERO）为真，则在返回这个页的地址之前将这个页填充满 0。

3：page_free()

```
    assert(pp->pp_ref == 0 || pp->pp_link == NULL);
    //panic if pp->pp_ref is nonzero or pp->pp_link is not NULL
    pp->pp_link = page_free_list;
    page_free_list = pp;
    //return a page to the free list
```

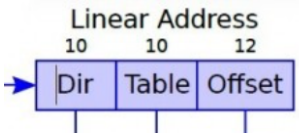
将 pp 对应的物理页释放掉。这里要检查一下(pp->pp_ref == 0 || pp->pp_link == NULL)是否成立，成立的话表示页已经不再被引用，才允许释放掉。

实验 3. [虚拟内存]
阅读材料完成 kern/pmap.c 中的下面几个子函数的编码：

1：pgdir_walk()

```
    int pdeIndex = (unsigned int)va >>22;
    //get the index in the page directory
    if(pgdir[pdeIndex] == 0 && create == 0)
        return NULL;
    //if the page doesn't exist and don't need to allocate a new one
(create==0), return NULL
    if(pgdir[pdeIndex] == 0)
    {//if the page doesn't exist and we need to allocate a new one
        struct PageInfo* page = page_alloc(1);
        if(page == NULL)
            return NULL;//the allocation fails
        page->pp_ref++;
        //the new page's reference count is incremented
        pte_t pgAddress = page2pa(page);
        pgAddress |= PTE_U;
        pgAddress |= PTE_P;
        pgAddress |= PTE_W;
        pgdir[pdeIndex] = pgAddress;
        //insert the new page table page into the page directory
    }
    //if the page exists, translate the address
    pte_t pgAdd = pgdir[pdeIndex];
    pgAdd = pgAdd>>12<<12;
    //get the page directory address
    int pteIndex =(pte_t)va >>12 & 0x3ff;
    //get the page table entry index
    pte_t * pte =(pte_t*) pgAdd + pteIndex;
    //get the physical address
    return KADDR( (pte_t) pte );
    //translate the physical address to virtual address
```

给定 pgdir,一个指向页目录基址的指针，pgdir_walk()函数返回线性地址 va 指向的页表项（PTE）。这需要从页目录到页表。如果对应的页表项不存在的话，如果参数 create 为 false，则直接返回；否则，在 pgdir 中为这个地址重新分配一个页。具体的，线性地址中前 10 位为页目录索引，中间 10 位为页表索引，后 12 位为页内偏移：



首先获得页目录索引。如果不存在且 create 为 false，则直接返回；如果 create 为 true 则利用 page_alloc()新分配一个页。如果页目录存在，获取页表的索引，就得到了页表项。最终返回物理地址。

2：boot_map_region()

```
//map [va, va+size) of virtual address space to physical [pa, pa+size)
while(size)
{
    pte_t* pte = pgdir_walk(pgdir, (void* )va, 1);
    //allocate new spaces
    if(pte == NULL)
        return;
    *pte= pa |perm|PTE_P;

    size -= PGSIZE;
    pa  += PGSIZE;
    va  += PGSIZE;
}
```

boot_map_region()函数将虚拟地址空间[va, va+size]映射到物理地址[pa, pa+size]。具体的，使用 pgdir_walk 函数查找 va 对应的页表项，没有就分配一个。对于每一个虚拟地址 va，使用 pgdir_walk，将页表项存储对应的 pa，完成映射。

3：page_lookup()

```
pte_t *pte = pgdir_walk(pgdir, va, 0);
//not allocating new spaces
if (!pte || !(*pte & PTE_P))
    return NULL;
//page not found
if (pte_store)
{
    //if pte_store is not zero, then we store in it
    //the address of the pte for this page
    *pte_store = pte;
    //found and set
}
return pa2page(PTE_ADDR(*pte));
```

返回映射到虚拟地址 va 的页。如果 pte_store 不为 0，就将这个页的页表项的存储在里面。使用宏 PTE_ADDR 获取页表项的地址，再使用宏 pa2page 获取对应的页。

4： page_remove()

```
    pte_t *pte;
    struct PageInfo *pp = page_lookup(pgdir, va, &pte);
    //find the requested page
    if (pp && (*pte & PTE_P))
    {
        page_decref(pp);
        //the ref count on the physical page should decrement
        *pte = 0;
        //the pg table entry corresponding to 'va' should be set to 0 in
case of errors
        tlb_invalidate(pgdir, va);
        //the TLB must be invalidated if remove an entry from the page table
    }
```

删除虚拟地址 va 对应的页。这个物理页的引用次数要减一，引用次数减为 0 时要释放掉；对应的页表项要设为 0；最终完成页的移除。

5： page_insert()

```
    pte_t *pte = pgdir_walk(pgdir, va, 1);
    //allocate new spaces
    if (pte == NULL)
        return -E_NO_MEM;
        //return -E_NO_MEM if page table couldn't be allocated

    pp->pp_ref++;
    //pp->pp_ref should be incremented if the insertion succeeds

    if (*pte & PTE_P)
    {
        page_remove(pgdir, va);
        //if there is already a page mapped at 'va', it should be
page_remove()
        pp->pp_link = NULL;
    }
    *pte = page2pa(pp) | perm | PTE_P;
    //permissions should be set to 'perm|PTE_P'
    return 0;
```

这是 JOS 在实现页面支持中最重要的一个函数，该函数的功能是将页面管理结构 pp 所对应的物理页面分配给线性地址 va；同时，将对应的页表项的 permission 设置成 PTE_P&perm。如果已经有页面映射到 va，则应该调用 page_remove 移除；如果需要的话，需要在 pgdir 中插入一个页表；插入成功的话，物理地址的引用 pp->pp_ref 要加一。

实验 4. [函数分析]

page_alloc 函数的作用是分配一个物理页并且返回一个 PageInfo 结构体的指针，请分析，PageInfo 中 pp_ref 的作用以及对编写 page_init()，page_alloc(),page_free()等函数的影响。

在应用中，由于虚拟内存的存在，可能存在着一个物理页（帧）被映射到多个进程的地址空间或者多个虚拟地址的情况。由于 struct PageInfo 对应着物理页，所以在其中加入 pp_ref 来记录对应的物理页被引用的次数。当 pp_ref 计数为 0 时，这个页就可以被释放掉了，因为没有进程在引用它。在实际中，计数 pp_ref 应该等于在所有页表中这个物理页出现在 UTOP 以下的次数（UTOP 以上的是内核启动时设置的，一般不会被释放掉，不用考虑）。

在 page_init()中，0 号物理页 pp_ref 设为 1，标记被使用，为实地址的 IDT 和 BIOS 结构预留了空间；其他正常分配的物理页 pp_ref 为 0，未被引用；[IOPHYSMEM,EXTPHYSMEM]段不能被分配，若出现在要求范围里，仅仅是将引用加 1。

在 page_alloc()中，引用次数不受影响，不改变 pp_ref，初始为 0。

在 page_free()中，要对 pp_ref 是否为 0（即这个页是否仍被引用）进行判断。如果仍被引用，就不能调用 page_free()释放这个页。