学号：5120809022
姓名：黄志强

Lab2 Memory Management

1. Main Idea

这个 lab 主要讲述的是 kernel virtual address 部分页表的建立，目前用户 virtual address 没有建立。那么如何建立？这个 lab 的想法比较简单，之前的 lab1 中我们建立了一个简单的页表，非常简单，就是将 virtual address-kernbase=physical address。这是一个线性的映射，为何我们要这样做？由于我们现在已经处于 protected mode，我们肯定是要用到页表的，所以在建立真正的页表之前，我们使用这个页表。有了这个页表，那么我们就可以建立自己的页表了，建立好了之后，将 cr3 设置成新的页表的 physical address 就完成了真正的页表。

页表的结构？我们先 alloc 一个 pde，这个一级页表，实际上，我们的一级页表只需要这一个 page 的 pde 就完全能够覆盖所有的物理空间。(virtual address 的分布，

```
// +--------10------+-------10-------+---------12----------+
// | Page Directory |     Page Table     | Offset within Page    |
// |          Index        |          Index        |                                |
// +----------------+----------------+----------------------+
```

一个 page 为 4K，pde 就有 4K/4=1K 个 pte 的 entry，正好就是前 10 位)，所以我们的 memory 的页表是一个一级页表，1024 个二级页表就可以了。由于我们是 page 对其的，所以我们的页表地址低 12 位应该为 0，这就来了一个技巧，我们就使用低 12 位来设置我们权限 bit (good idea ! !)。

页表的建立过程？在一级页表中（kern_pgdir）根据 VA 前 10bit 找到二级页表的 entry，那么就存在以下情况？1）页表 entry 是 0，那么我们分配一个物理 page 来作为二级页表，将他的地址给 entry 并设置权限；2）页表 entry 不为 0，但是 PET_P 为 0，代表不再内存中，操作如 1）;3) 页表 entry 存在，但权限不够 (由于我们是 kernel page，所以不存在这个情况)，返回 NULL；4）页表存在，而且权限可以，那么我们直接找到二级页表就可以了。找到二级页表，通过 PTX（VA）+*PDE 找到二级页表中的 entry，重复上面的情况就可以了。

基本想法有了，下面介绍这个 lab 的详细情况

2. Alloc Physical Address

这个部分是为页表的建立做好前提-物理页的分配。我们在这一部分就是完成分配物理页的功能，以下就是我们需要实现的接口：

`boot_alloc()`
`page_init()`
`page_alloc()`
`page_free()`
page_alloc_npages(int alloc_flags, int n)

page_free_npages(struct Page *pp, int n)
page_realloc_npages(struct Page *pp, int old_n, int new_n);

下面我们来一一介绍这些函数的功能和实现。

1) boot_alloc()

功能：在页表建立之前，分配 VA 给用户。

实现：

```
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
    result = nextfree;
    if(n>0)
    {
        nextfree = ROUNDUP(nextfree+n,PGSIZE);
        if((int)(result + n) >= 0xffffffff)
        {
            panic("out of memory\n");
            return NULL;
        }
    }
    return result;
}
```

Nextfree 初始化就是 kernel elf 文件格式 bss 后面，就是空闲空间的第一个 byte，在 kernbase 和 nextfree 之间是 allocated 的 memory，nextfree 之后才是 free memory。那么代码很简单，只需要返回 nextfree，移动 nextfree 并且对齐就可以了，代码比较简单，这里不再详细介绍。调用的地方主要有两处：

```
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
pages = (struct Page*)boot_alloc(npages*sizeof(struct Page));
```

在这里我不得不说的是，本来pages实在kern_pgdir后一个page的，但是后面我们改变了pages的位置，后面再说了。

2) page_init()

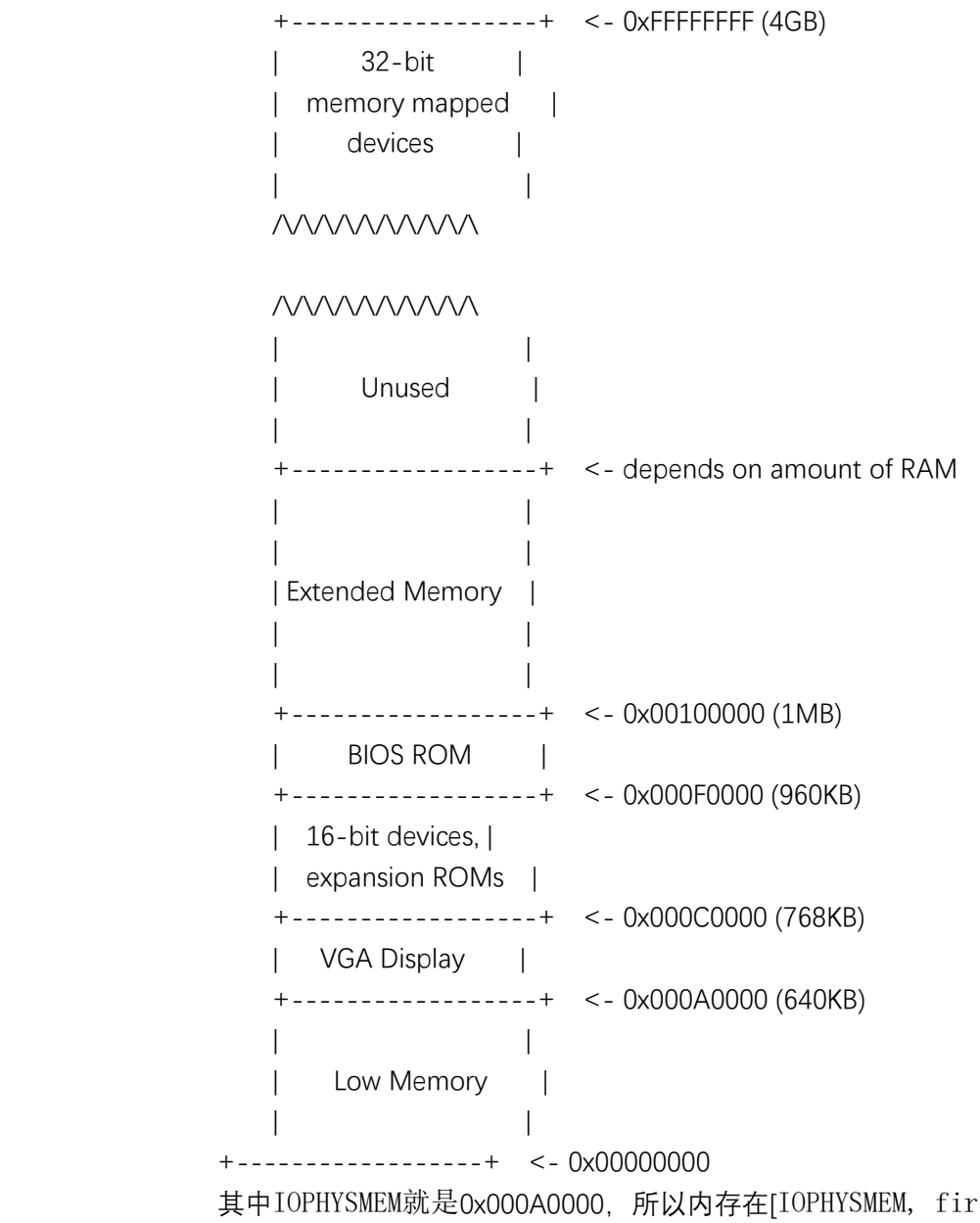
功能：初始化page，将free page用链表连起来。

代码：

```
physaddr_t firstFreePhysAddr=(physaddr_t)PADDR(boot_alloc(0));
page_free_list=NULL;
for (i = npages-1; i >= 0; i--)
{
     physaddr_t pagePhysAddr=(physaddr_t)PGADDR(0,i,0);
     if(pagePhysAddr==0||
```

```
    (pagePhysAddr>=IOPHYSMEM&&pagePhysAddr<firstFreePhysAddr))
     {
    pages[i].pp_ref=1;
    pages[i].pp_link=NULL;
     }
     else
     {
    pages[i].pp_ref=0;
    pages[i].pp_link=page_free_list;
    page_free_list=&pages[i];
         }
}
```

boot_alloc(0)得到的就是nextfree，在firstFreePhysAddr后面的才是free，当然我们观察PA的结构就可以看到：



其中IOPHYSMEM就是0x000A0000，所以内存在[IOPHYSMEM，firstFreePhysAddr]

之间的就是allocated的，其他是free。还有比较trick的一点就是，从npages-1到0倒过来，这样做就可以使得page_free_list指在pages的head，并且free list是由低地址向高地址排列的，代码比较好理解。

3） page_alloc()

功能：分配一个page。

代码：

```
struct Page* allocPage;
if(page_free_list==NULL)
    return NULL;
allocPage=page_free_list;
page_free_list=page_free_list->pp_link;
allocPage->pp_link=NULL;
if(alloc_flags&&ALLOC_ZERO)
{
    void* kvirAddr=page2kva(allocPage);
    memset(kvirAddr,0,PGSIZE);
}
return allocPage;
```

很简单，只要返回free list的head，然后向下移动就可以了。代码

```
if(alloc_flags&&ALLOC_ZERO)
{
    void* kvirAddr=page2kva(allocPage);
    memset(kvirAddr,0,PGSIZE);
}
```

只是简单的设置为0，当然因为memset是VA，所以先转化。

4） page_free()

功能：free一个page

代码：

```
if(page_free_list==NULL)        //代码这个 page 就是 head
{
    pp->pp_link=page_free_list;
    page_free_list=pp;
    return;
}
if(page2pa(pp)<page2pa(page_free_list)) //page 应该放在 head 之前
{
    pp->pp_link=page_free_list;
    page_free_list=pp;
    return;
}
struct Page* skip=page_free_list;
int found=0;  //标记是否找到了 page 的位置
while(skip->pp_link!=NULL)    //找到 page 在 free list 中应该处在的位置
{
```

```
            if(page2pa(pp)>page2pa(skip)&&
               page2pa(pp)<page2pa(pp->pp_link))
            {
                found=1;
                break;
            }
            skip=skip->pp_link;
        }
        if(found==0)      //没找到 page 的位置，那么放到 list 的 tail 就可以了
        {
            skip->pp_link=pp;
            pp->pp_link=NULL;
            return;
        }
        if(found==1)  //找到了，进行插入就可以
        {
            struct Page* temp=skip->pp_link;
            skip->pp_link=pp;
            pp->pp_link=temp;
        }
```

将page加入free list就可以了，但是我实现的是保证了page的位置。使得free list有低地址向高地址增加，代码的理解请看注释。

5） page_alloc_npages(int alloc_flags, int n)

功能：分配 n 个连续的 page

代码：

```
    struct Page* allocPage=page_free_list;
     int found=0;     //判断是不是有连续的地址
     while((found=check_continuous(allocPage,n))!=1)
     {
         allocPage=allocPage->pp_link; //找到这个地址
     }
     if(found==0)
         return NULL;
     if(allocPage==page_free_list)     //这个地址是 list 的 head
     {
         struct Page* after=allocPage;
         struct Page* tail=NULL;
         int i;
         for(i=0;i<n;i++)
         {
             if(alloc_flags & ALLOC_ZERO)
             {
              memset(page2kva(after),0,PGSIZE);
                         }
```

```
            tail=after;          //连续地址最后一个 page
            after=after->pp_link;   //连续地址后一个 page
        }
        page_free_list=after;     //移动 list 的 head
        tail->pp_link=NULL;
    }
    Else                              //这个地址不是 list 的 head
    {
        struct Page* prev=page_free_list;
        while(prev->pp_link!=allocPage)   //找到连续地址前面一个 page
            prev=prev->pp_link;
        struct Page* after=allocPage;
        struct Page* tail=NULL;
        int i;
        for(i=0;i<n;i++)
        {
            if(alloc_flags & ALLOC_ZERO)
            {
              memset(page2kva(after),0,PGSIZE);
                        }
            tail=after;     //连续地址最后一个 page
            after=after->pp_link;     //连续地址后一个 page
        }
        prev->pp_link=tail->pp_link;   //前后连起来
        tail->pp_link=NULL;
    }
    return allocPage;
  }
```

逻辑很简单，先看看是否有连续的，有的话就找到连续地址的起始点，然后返回就可以，理解看代码。

6） page_free_npages()

功能：free连续n个page

代码：

```
  for(i=0;i<n;i++)
    {
        skip=pp;
        pp=pp->pp_link;
        page_free(skip);
    }
```

一个个free就可以了，注意要先获取page在移动，不然出错。

7） page_realloc_npages()

功能：If new n is smaller than the old n, you can free the last pages. If new n is larger than the old n, you should first check whether the following pages are free. If not, you can do it in simple way. Otherwise, allocate these pages.

代码：

```
if(old_n==new_n)        //不需要做任何改变
        return pp;
 if(new_n<old_n)       //小，free 后面的就可以了
 {
        struct Page* skip=pp;
        int i;
        for(i=0;i<new_n;i++)
              skip=skip->pp_link;   //找到要 free 的第一个 page
        struct Page* temp;
        for(i=new_n;i<old_n;i++) //一个个 free
        {
              temp=skip;
              skip=skip->pp_link;
              page_free(temp);
        }
        return pp;
 }
 if(new_n>old_n)   //大，看看后面有连续的吗
 {

        struct Page* tail=pp;
        int i=0;
        for(i=0;i<old_n-1;i++)
              tail=tail->pp_link; //找到后面的第一个 page

        struct Page* skip=page_free_list;
        int found=0;int following=0;
        while(skip)
        {
              if(page2pa(skip)-page2pa(tail)==PGSIZE)//判断后面第一个 page free
              {
                    found=1;
                    break;
              }
              skip=skip->pp_link;
        }
        if(found==1)   //确实是 free 的
        {
              if(check_continuous(skip,new_n-old_n)==1)//看是否有连续的 page
                  following=1;
        }
        if(following==1) //有连续的 page，分配后面的
        {
```

```
                struct Page* prev; //page 前面一个 page
                struct Page* after; //page 后面一个
                struct Page* ext=NULL; //要多加的 page
                if(skip==page_free_list) //如果是 head
                {
                        after=skip;
                        for(i=0;i<new_n-old_n;i++)
                        {
                        ext=after;   //new_n 的最后一个 page
                        after=after->pp_link; //后面一个 page
                                        }
                        ext->pp_link=NULL;
                        page_free_list=after; //free list 移动到后面就可以了
                }
                else //不是 head
                {
                        prev=page_free_list;
                        while(prev->pp_link!=skip)
                        prev=prev->pp_link;   //前面一个 page

                        after=skip;
                        for(i=0;i<new_n-old_n;i++)
                        {
                        ext=after; //new_n 的最后一个 page
                        after=after->pp_link;//后面一个 page
                                        }
                        ext->pp_link=NULL;
                        prev->pp_link=after; //free list  前后连接起来
                }
                return pp;
        }
        else //如果后面没有，先 free，在 alloc 就可以了
        {
                struct Page* allocPage=page_alloc_npages(1,new_n);
                if(allocPage==NULL)
                        return NULL;
                memmove(page2kva(allocPage),page2kva(pp),old_n*PGSIZE);
                page_free_npages(pp,old_n);
                return allocPage;
        }
    }
```

分情况写就可以了，代码看注释，memmove是用来复制内存值得，传的是指针，所以先转成VA。

3. Page Table

前面已经讲述关于物理分配的核心的函数，我们现在来看看他们是怎么运作的，首先我们看看mem_init()做了些什么：

```
boot_map_region(kern_pgdir,UPAGES,ROUNDUP(npages*sizeof(structPage),PGSIZE),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,(~KERNBASE)+1,
 0,PTE_W|PTE_P);
```

这个函数主要是建立了一个页表，第一个将 pages 映射到 VA 的 UPAGES 这个位置，这就是我们先前讲到 pages 不再 kern_pgdir 后面的原因。第二个将 KSTACKTOP (kernel 的 stack）映射到 bootstack。第三个将 kernelbase 上面的映射到 0 地址。这样建立了关于 kernel 的三个区域的页表之后，直接 lcr3(PADDR(kern_pgdir));切换页表，这样就完成 kernel 页表的建立。这涉及到下面一个核心函数：

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

下面一一讲解：

1） pgdir_walk();

功能：找到 VA 对应在页表中的 pte，不存在就建立页表。

代码：

```
struct Page* newPage;
  pde_t* pde=pgdir+PDX(va);
  pte_t* pte;
  if(*pde&PTE_P)   //pde 中有相应的 entry
  {
        pte=(pte_t*)KADDR(PTE_ADDR(*pde));
        return pte+PTX(va);   //返回 pte
  }
  if(create==0)   //没有 entry 并且不创立
  {
        return NULL;
  }
  else
  {
        newPage=page_alloc(ALLOC_ZERO);//创立一个二级 page
        if(newPage==NULL)
              return NULL;
        newPage->pp_ref=(newPage->pp_ref)+1;
        *pde=page2pa(newPage)|PTE_P|PTE_W|PTE_U; //设置 pde 中的 entry
        pte=(pte_t*)KADDR(PTE_ADDR(*pde));
        return pte+PTX(va); //返回 pte
```

```
        }
```

2） boot_map_region()

功能：将一块 VA 映射到一块 PA

代码：

```
    int n=size/PGSIZE;
      int i;
      pte_t* pte;
      for(i=0;i<n;i++)
      {
            pte=pgdir_walk(pgdir,(void*)va,1);//找到 pte，没有就创建
            *pte=pa|perm|PTE_P;     //将 pte 设置成相应的 pa 就可以了
            pa+=PGSIZE;
            va+=PGSIZE;
      }
```

一个一个 page 映射就可以。

3） page_lookup()

功能：找到 va 对应的 page

代码：

```
    pte_t *pte=pgdir_walk(pgdir,va,0); //找到 PTE
      if(pte_store)
            *pte_store=pte;
      if((pte!=NULL)&&(*pte&PTE_P))
            return pa2page(PTE_ADDR(*pte)); //取出 pte 的值转化成 page 就可以了
      return NULL;
```

比较简单，找 page table 存的值就可以。

4） page_remove()

功能：删除 VA 对应的 page 就可以了

代码：

```
    pte_t* pte;
      struct Page* res;
      res=page_lookup(pgdir,va,&pte);   //找到 page
      if(res!=NULL)
      {
            page_decref(res);   //ref 减一，ref 为 0，就会 free
            *pte=0;                 //page table 的 entry 设置为 0；
            tlb_invalidate(pgdir,va);//tlb 中删除
      }
```

找到 VA 的 page，free 就可以了。

5） page_insert()

功能：将某一个 page 插入到某个 VA

代码：

```
    pte_t* pte=pgdir_walk(pgdir,va,1);   //找到 pte
      if(pte==NULL)
      {
```

```
        return -E_NO_MEM;
    }
    if(*pte&PTE_P)     //pte 对应的 page 存在
    {
        if(PTE_ADDR(*pte)==page2pa(pp)) //就是当前的 page
        {
            tlb_invalidate(pgdir,va);    //tlb 删除
            pp->pp_ref=(pp->pp_ref)-1; //ref-1
        }
        else //不是当前 page
        {
            page_remove(pgdir,va);    //删除 page
        }
    }
    *pte=page2pa(pp)|perm|PTE_P;     //将 pte 设置为当前 page 的 PA
    pp->pp_ref=(pp->pp_ref)+1;       //ref+1
    return 0;
```

看代码。

4. 总结

总的来说，这个 lab 的逻辑比上一个 lab 要清晰的多，非常好理解，直到他要干什么，也知道怎么干，代码量比较少也比较简单，是一个 good lab。