JOS LAB2：Memory Management

JOS Lab2 要求我们实现操作系统的内存管理，主要包括两个大的方面，一是为kernel实现物理内存的分配功能，物理内存的分配以Page为单位，大小为4096Bytes，我们的任务是实现空闲Page的管理和使用时的分配。一是实现虚拟地址到物理地址的映射。

已实现Chanllenges，请看本文档最后，questions的回答请见answer-lab2.txt

Part1: Physical Page Management

Part1的任务主要是完成`boot_alloc()`、`mem_init()`、`page_init()`、`page_alloc()`、`page_free()`五个函数，实现对于空闲Page的管理、分配及回收。

boot_alloc()

boot_alloc函数只在JOS创建内存管理系统时使用，负责开辟出n个Byte的空闲空间，并返回这段空闲空间的首地址。从JOS原本给出的代码和注释可以得知end以上均为可用地址，nextfree指向下一块可用空间的起始地址，我们只需将nextfree初始化为end的4K字节对齐，然后加上我们要分配的空间大小，并返回初始时的nextfree即可。

```
static void * boot_alloc(uint32_t n)
{
    static char *nextfree;
    char *result;

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

    result=nextfree;
    nextfree+=ROUNDUP(n,PGSIZE);
    return result;
}
```

mem_init()

Part1里mem_init函数的改动比较少，只需要调用boot_alloc函数来对物理页数组pages进行初始化即可，参数为内存中物理页数目npages乘以单个Page的大小

```
panic("mem_init: This function is not finished\n");
```

同时需要将`panic("mem_init: This function is not finished\n");`注释掉，以免为后面测试带来干扰。

page_init()

page_init函数负责对pages数组中的每个struct Page(此struct Page为数据结构，而非物理页Page，可以通过page2kva函数得到对应的物理页Page的虚拟地址)以及page_free_list(记录了所有仍空闲的Page)进行初始化，但并不是所有pages中的Page都能被记录在page_free_list中，我们需要将一些系统已经用到的页面从中剔除出去，包括0地址上的第一个页面（包括了IDT），IO Hole（包括了VGA等）、Kernel及kern_pgdir、pages。0地址的起始地址就是0，空间大小为一个Page，IO Hole的所占地址空间是`[IOPHYSMEM, EXTPHYSMEM)`，均为物理地址，Kernel部分的起始地址正好是`EXTPHYSMEM`，结束地址为`end-KERNBASE+PGSIZE+npages*sizeof(struct Page)`，即Kernel结束地址加上kern_pgdir加上pages所占的空间，之所以Kernel结束地址要用`end-KERNBASE`，是因为end是物理地址，要转化为虚拟地址。

```
void page_init(void)
{
    extern char end[];
    size_t i;
    size_t hole_begin=PGNUM(IOPHYSMEM);
    size_t hole_end=PGNUM(end-KERNBASE+PGSIZE+npages*sizeof(struct Page));
    for (i = 0; i < npages; i++)
    {
```

```
        if(i==0||(hole_begin<=i&&i<=hole_end))
        {
            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];
        }
    }
    chunk_list = NULL;
}
```

page_alloc()

page_alloc函数负责分配空闲Page，即在page_free_list中取出一个空闲Page，将其从page_free_list中剔除出去并作为分配结果返回，实现很简单，将page_free_list的第一个元素取出，变page_free_list为其pp_link，如果page_free_list为空，即没有空闲Page，就返回NULL。同时，还要按照alloc_flags参数的要求决定是否要对被分配的Page进行初始化。

```
struct Page * page_alloc(int alloc_flags)
{
    if(page_free_list==NULL)return NULL;
    struct Page* page1=page_free_list;
    page_free_list=page_free_list->pp_link;
    if(alloc_flags&ALLOC_ZERO)memset(page2kva(page1),0,PGSIZE);
    return page1;
}
```

page_free()

page_free函数负责回收Page，将其重新记录在page_free_list中。

```
void page_free(struct Page *pp)
{
    if(pp->pp_ref==0)
    {
        pp->pp_link=page_free_list;
        page_free_list=pp;
    }
}
```

Part 2: Virtual Memory

Part2让我们进一步实现内存管理，主要是完成`pgdir_walk()`、`boot_map_region()`、`page_lookup()`、`page_remove()`、`page_insert()`五个函数，实现虚拟-物理地址映射。

pgdir_walk()

pgdir_walk函数根据参数虚拟地址va，查找其对应在二级页表中的page table entry（PTE），并返回指向此PTE的指针，如果暂时没有此PTE，则根据create参数的指示选择是否创建这一PTE。查找的方式就是从一级页表中获取va对应的第PDX(va)项页目录，从而找到对应的二级页表，从二级页表中获取va对应的第PTX(va)项页目录，即找到了要返回的PTE。

```
pte_t * pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    pte_t* pte1=&pgdir[PDX(va)];
    if((*pte1)&PTE_P)
    {
        pte_t* pte2=KADDR(PTE_ADDR(*pte1));
        return &pte2[PTX(va)];
    }
    else
```

```
    {
        if(create==1)
        {
            struct Page* page1=page_alloc(1);
            if(page1==NULL)return NULL;
            page1->pp_ref=1;
            pgdir[PDX(va)]=page2pa(page1)|PTE_P|PTE_W|PTE_U;
            pte1=page2kva(page1);
            return &pte1[PTX(va)];
        }
        else
        {
            return NULL;
        }
    }
    return NULL;
}
```

boot_map_region()

boot_map_region函数将虚拟地址空间[va, va+size)映射到物理地址空间[pa, pa+size)，并将每一个对应的页表项的permission设置为perm|PTE_P,可以借助create参数为1的pgdir_walk函数完成。

```
static void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    ROUNDUP(size,PGSIZE);
    int pagenum=size/PGSIZE;
    int i=0;
    for(;i<pagenum;++i)
    {
        pte_t* pte1=pgdir_walk(pgdir,(void*)va,1);
        if(pte1==NULL)return;
        *pte1=pa|perm|PTE_P;
        pa+=PGSIZE;
        va+=PGSIZE;
    }
}
```

page_lookup()

page_lookup函数查找并返回参数va对应的物理页，同时如果pte_store参数不为空，则将找到的pte放到pte_store指向的地址中。查找的方式是使用create参数为0的pgdir_walk函数，如果返回为NULL，说明没有找到。

```
struct Page * page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t* pte1=pgdir_walk(pgdir,va,0);
    if(pte_store!=0)*pte_store=pte1;
    if((pte1!=NULL)&&(((*pte1)&PTE_P)!=0))return pa2page(PTE_ADDR(*pte1));
    return NULL;
}
```

page_remove()

page_remove函数取消参数va与对应物理页的映射，首先通过page_lookup函数查找va对应的Page，如果返回为NULL，那么不用进行处理，否则要对返回的Page执行refrence-1操作，如果减一后reference变为0，则要回收此Page（page_decref函数已帮忙实现），最后通知tlb高速缓存这一缓存失效。

```
void page_remove(pde_t *pgdir, void *va)
{
    pte_t* pte1=NULL;
    struct Page* page1=page_lookup(pgdir,va,&pte1);
    if(page1==NULL)return;
    page_decref(page1);
    *pte1=0;
```

```
    tlb_invalidate(pgdir,va);
}
```

page_insert()

page_insert函数将参数虚拟地址va映射到参数`struct Page* pp`所对应的物理页，实现方式为查找va对应的pte，然后将va对应的pte内容更改为pp对应的物理页的物理地址，并将pp对应的Page的reference加一，va本来映射的Page的reference减一。这里还要特别处理一下va本来映射的Page就是pp对应的Page的情况，即只需要按perm参数更改其permission即可。

```
int page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    pte_t* pte1=pgdir_walk(pgdir,va,1);
    if(pte1==NULL)return -E_NO_MEM;
    struct Page* page1=page_lookup(pgdir,va,NULL);
    if(page1==NULL)
    {
        *pte1=page2pa(pp)|perm|PTE_P;
        pp->pp_ref+=1;
    }
    else if(page1==pp)
    {
        *pte1=page2pa(pp)|perm|PTE_P;
    }
    else
    {
        page_remove(pgdir,va);
        *pte1=page2pa(pp)|perm|PTE_P;
        pp->pp_ref+=1;
    }
    return 0;
}
```

Part3

Part3需要我们进行初始化Kernel的页目录、页表，加载至cr3等操作。主要是按注释提示借助Part1/2已经实现的代码修改mem_init()函数。

1

首先是将pages映射到UPAGES-UVPT地址空间，perm为PTE_U|PTE_P

```
boot_map_region(kern_pgdir,UPAGES,ROUNDUP(npages * sizeof(struct Page),PGSIZE),PADDR(pages),PTE_U|PTE_P);
```

然后是映射bootstack到Kernel的stack,perm为PTE_W | PTE_P

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

2

这里我们要完成boot_map_region_large函数，boot_map_region_large和boot_map_region函数不同点，我的理解是前者映射更大的物理空间，修改的是一级页表，后者影射的是较小快的物理空间，修改的是二级页表。

```
static void boot_map_region_large(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    ROUNDUP(size,PTSIZE);
    int ptnum=size/PTSIZE;
    int i=0;
    for(;i<ptnum;++i)
    {
        pgdir[PDX(va)]=pa|perm|PTE_P|PTE_PS;
        pa+=PTSIZE;
        va+=PTSIZE;
    }
}
```

然后我们需要对mem_init()再次修改，用boot_map_region_large函数将从KERNBASE到0xffffffff这一段内存空间映射到从0开始的物理地址（mapping anyway）。要用到PDE的PS位，所以开启CR4的PSE，使每个PDE表中的entry对应4MB的内存。

```
    uint32_t cr4;
    cr4 = rcr4();
    cr4 |= CR4_PSE;
    lcr4(cr4);
    boot_map_region_large(kern_pgdir, KERNBASE, ~KERNBASE + 1, 0, PTE_W | PTE_P);
```

Challenges

Chanllenge1

showmappings

我实现了mon_showmappings函数，将bengin_addr到end_addr虚拟地址对应的所有页表展示出来。实现方式是通过pgdir_walk函数获取所有这一空间对应到的pte，然后获取pte中的物理地址对应的Page即permissions。

```
uint32_t strtoint(char* str)
{
    if(str==NULL)return 0;
    if((*str=='\0')||(*(str+1)=='\0'))return 0;
    char* buf=str+2;//start after 0x
    uint32_t result=0;
    while((*buf)!='\0')
    {
        uint32_t num=0;
        if((*buf>='a')&&(*buf<='f'))
        {
            num=*buf-'a'+10;
        }
        else if((*buf>='A')&&(*buf<='F'))
        {
            num=*buf-'A'+10;
        }
        else if((*buf>='0')&&(*buf<='9'))
        {
            num=*buf-'0';
        }
        result=result*16+num;
        buf+=1;
    }
    return result;
}
```

```
int mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    if (argc !=3 ) {
        cprintf("Usage: showmappings 0xbegin_addr 0xend_addr\n");
        return 0;
    }
    uint32_t begin_addr=strtoint(argv[1]);
    uint32_t end_addr=strtoint(argv[2]);
    for(;begin_addr<=end_addr;begin_addr+=PGSIZE)
    {
        pte_t* pte1=pgdir_walk(kern_pgdir,(void*)begin_addr,1);
        if(pte1==NULL)cprintf("Showmappings: memory not enough\n");
        else if((*pte1)&PTE_P)
        {
            cprintf("0x%x :physical mapping page ox%x,permission PTE_P %d PTE_W %d PTE_U %d\n" ,begin_addr,*pte1,(*pte1)&PTE_P, ((
        }
    }
```

```
    return 0;
}
```

Explicitly set, clear, or change permissions

我实现了一个mon_changeperm函数，用于更改permissions，实现方式同样是借助pgdir_walk函数获得虚拟地址va对应的pte，将pte的permissions更改为我们想要的形式。

```
int mon_changeperm(int argc, char **argv, struct Trapframe *tf)
{
    if (argc <= 3) {
        cprintf("Usage: changeperm 0xaddr [commandtype] [permtype] [permvalue]\n");
        cprintf("[commandtype]:0/1/2    0 represents set, 1 represents change, 2 represents clear\n");
        cprintf("[permtype]:0/1/2    0 represents PTE_P, 1 represents PTE_W, 2 represents PTE_U\n");
        cprintf("[permvalue]:0/1    be null if command type is clear\n");
        return 0;
    }
    uint32_t addr=strtoint(argv[1]);
    uint32_t commandtype=0;
    char* str1=argv[2];
    if((*str1>='0')&&(*str1<='2')&&(*(str1+1)=='\0'))commandtype=*str1-'0';
    else
    {
        cprintf("Usage: changeperm 0xaddr [commandtype] [permtype] [permvalue]\n");
        cprintf("[commandtype]:0/1/2    0 represents set, 1 represents change, 2 represents clear\n");
        cprintf("[permtype]:0/1/2    0 represents PTE_P, 1 represents PTE_W, 2 represents PTE_U\n");
        cprintf("[permvalue]:0/1    be null if command type is clear\n");
        return 0;
    }
    if(((commandtype==0||commandtype==1)&&argc!=5)||(commandtype==2&&argc!=4))
    {
        cprintf("Usage: changeperm 0xaddr [commandtype] [permtype] [permvalue]\n");
        cprintf("[commandtype]:0/1/2    0 represents set, 1 represents change, 2 represents clear\n");
        cprintf("[permtype]:0/1/2    0 represents PTE_P, 1 represents PTE_W, 2 represents PTE_U\n");
        cprintf("[permvalue]:0/1    be null if command type is clear\n");
        return 0;
    }
    uint32_t permtype=0;
    char* str2=argv[3];
    if(*str2>='0'&&*str2<='2'&&*(str2+1)=='\0')permtype=*str2-'0';
    else
    {
        cprintf("Usage: changeperm 0xaddr [commandtype] [permtype] [permvalue]\n");
        cprintf("[commandtype]:0/1/2    0 represents set, 1 represents change, 2 represents clear\n");
        cprintf("[permtype]:0/1/2    0 represents PTE_P, 1 represents PTE_W, 2 represents PTE_U\n");
        cprintf("[permvalue]:0/1    be null if command type is clear\n");
        return 0;
    }
    uint32_t permvalue=0;
    if(commandtype!=2)
    {
        char* str3=argv[4];
        if(*str3>='0'&&*str3<='2'&&*(str3+1)=='\0')permvalue=*str3-'0';
        else
        {
            cprintf("Usage: changeperm 0xaddr [commandtype] [permtype] [permvalue]\n");
            cprintf("[commandtype]:0/1/2    0 represents set, 1 represents change, 2 represents clear\n");
            cprintf("[permtype]:0/1/2    0 represents PTE_P, 1 represents PTE_W, 2 represents PTE_U\n");
            cprintf("[permvalue]:0/1    be null if command type is clear\n");
            return 0;
        }
    }
    pte_t* pte1=pgdir_walk(kern_pgdir,(void*)addr,1);
    cprintf("Before: 0x%x :permission PTE_P %d PTE_W %d PTE_U %d\n",addr,(*pte1)&PTE_P, ((*pte1)&PTE_W)>>1, ((*pte1)&PTE_U)>>2);
    if(pte1==NULL)return 0;
```

```
    uint32_t perm=0;
    if(permtype==0)perm=PTE_P;
    if(permtype==1)perm=PTE_W;
    if(permtype==2)perm=PTE_U;
    if(permvalue==1)*pte1=*pte1|perm;
    else *pte1=*pte1&(~perm);
    if(commandtype==2)*pte1=*pte1&(~perm);
    cprintf("After: 0x%x :permission PTE_P %d PTE_W %d PTE_U %d\n",addr,(*pte1)&PTE_P, ((*pte1)&PTE_W)>>1, ((*pte1)&PTE_U)>>2);
    return 0;
}
```

Dump contents

我实现了mon_dumpmem函数，将begin_addr到end_addr地址空间上的内容打印出来。begin_addr和end_addr可以是虚拟地址，也可以是物理地址。如果是物理地址，则要借助PGNUM宏来检查确定没有超过npages，然后转化为虚拟地址。如果是虚拟地址，则用create参数位0的pgdir_walk来检查是否已映射，如果已映射则输出地址上的值，否则返回NULL。

```
int mon_dumpmem(int argc, char **argv, struct Trapframe *tf)
{
    if (argc !=4 )
    {
        cprintf("Usage: dumpmem [ADDR_TYPE] 0xbegin_addr 0xend_addr\n");
        cprintf("[ADDR_TYPE]:P/V    P represents physicval address type,V represents virtual address type\n");
        return 0;
    }
    char type='0';
    if((*(argv[1]))=='P'||(*(argv[1]))=='V')
        type=(*(argv[1]));
    else
    {
        cprintf("Usage: dumpmem [ADDR_TYPE] 0xbegin_addr 0xend_addr\n");
        cprintf("[ADDR_TYPE]:P/V    P represents physicval address type,V represents virtual address type\n");
        return 0;
    }
    uint32_t begin_addr=strtoint(argv[2]);
    uint32_t end_addr=strtoint(argv[3]);
    if(begin_addr>=end_addr||(begin_addr%4)!=0||(end_addr%4)!=0)
    {
        cprintf("Invalid addresses\n");
        return 0;
    }
    if(type=='P')
    {
        if(PGNUM(begin_addr) >= npages||PGNUM(end_addr) >= npages)
        {
            cprintf("Invalid addresses\n");
            return 0;
        }
        begin_addr=(uint32_t)KADDR(((physaddr_t)begin_addr));
        end_addr=(uint32_t)KADDR(((physaddr_t)end_addr));
    }
    while(begin_addr<=end_addr)
    {
        pte_t* pte1=pgdir_walk(kern_pgdir,(void*)begin_addr,0);
        if(pte1==NULL)cprintf("0x%08lx:NULL\n",begin_addr);
        else cprintf("0x%08lx:0x%x\n",begin_addr,*((uint32_t*)(begin_addr)));
        begin_addr+=4;
    }

    return 0;
}
```

Chanllenge2

这个chanllenge要我们实现`struct Page *alloc_page_with_color(int alloc_flags, int color)`函数，这个函数分配具有特定`color`的Page并返

回。
对某一特定Page，color的标定是其物理地址的第12和第13个bit（第0到第11bit为page offset）。我们在page_free_list中查找符合条件的Page即可。

```
struct Page *alloc_page_with_color(int alloc_flags, int color)
{
    if(page_free_list==NULL)return NULL;
    struct Page* page1=page_free_list;
    physaddr_t addr=page2pa(page1);
    if((((uint32_t)addr)&4)==color)
    {
        page_free_list=page_free_list->pp_link;
        if(alloc_flags&ALLOC_ZERO)memset(page2kva(page1),0,PGSIZE);
        return page1;
    }
    while(page1->pp_link!=NULL)
    {
        physaddr_t addr=page2pa(page1->pp_link);
        if((((uint32_t)addr)&4)==color)
        {
            if(alloc_flags&ALLOC_ZERO)memset(page2kva(page1->pp_link),0,PGSIZE);
            page1->pp_link=page1->pp_link->pp_link;
            return page1;
        }
        page1=page1->pp_link;
    }
    return NULL;
}
```

至此，JOS Lab2全部完成。