Lab2 Report

朱新宇 1120379029

CSDI

2013年4月12日

JOS Lab2 Report

Part1. 物理内存的页管理

在part1中主要是通过pages来实现对物理内存的管理，操作系统通过pages结构可以以页为单位标记物理页的状态.

```
struct Page {
        // Next page on the free list.
        struct Page *pp_link;

        // pp_ref is the count of pointers (usually in page table entries)
        // to this page, for pages allocated using page_alloc.
        // Pages allocated at boot time using pmap.c's
       // boot_alloc do not have valid reference count fields.
        uint16_t pp_ref;
};
```

连续的Page数组线性代表了连续的物理页，pmap.h 提供了page2pa, pa2page和page2kva三个函数来处理page，在page2pa中通过传入的page和page的第一个元素地址的差值计算page对应的物理地址，再通过KADDR宏就可以转成kernel virtual address。

存放pages选择放在kernel的后面，也就是通过链接器提供的end获取链接的最后面的位置，然后分配相应的空间。

在实现上则是通过boot_alloc分配npages*sizeof(struct Page)的空间

```
static void *
boot_alloc(uint32_t n)
{
      static char *nextfree;  // virtual address of next byte of free mem-
ory
      char *result;

      // Initialize nextfree if this is the first time.
      // 'end' is a magic symbol automatically generated by the linker,
      // which points to the end of the kernel's bss segment:
      // the first virtual address that the linker did *not* assign
      // to any kernel code or global variables.
      if (!nextfree) {
            extern char end[];
            nextfree = ROUNDUP((char *) end, PGSIZE);
      }

      // Allocate a chunk large enough to hold 'n' bytes, then update
      // nextfree.  Make sure nextfree is kept aligned
      // to a multiple of PGSIZE.
      //
      // LAB 2: Your code here.

      if( n == 0 )
```

```
                return (void *)nextfree;

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

在分配完空间后需要调用page_init初始化pages，在初始化时需要做的是标记已经被使用的page，然后用page_free_list连接所有空闲的page，我采用从后向前初始化，由于kernel以及pages等已经占用了一部分物理页，所以需要通过boot_alloc（0）来获取kernel代码空间的最后,从这里到extended memory的开始都是已经被占用的，此外IO hole和第一个页，完成后page_free_list指向第一个空闲的page。

```
void
page_init(void)
{
    // The example code here marks all physical pages as free.
    // However this is not truly the case.  What memory is free?
    //  1) Mark physical page 0 as in use.
    //     This way we preserve the real-mode IDT and BIOS structures
    //     in case we ever need them.  (Currently we don't, but...)
    //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
    //     is free.
    //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
    //     never be allocated.
    //  4) Then extended memory [EXTPHYSMEM, ...).
    //     Some of it is in use, some is free. Where is the kernel
    //     in physical memory?  Which pages are already in use for
    //     page tables and other data structures?
    //
    // Change the code to reflect this.
    // NB: DO NOT actually touch the physical memory corresponding to
    // free pages!
    size_t i;
    int cnt;
    uint32_t add;
    size_t mile = PADDR(boot_alloc(0)) / PGSIZE;
    page_free_list = NULL;
    pages[0].pp_ref = 1;
    for(i = npages - 1; i >= mile; --i){
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    for(i = npages_basemem - 1; i >= 1; --i){
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    for(i = npages_basemem; i < mile; ++i)
        pages[i].pp_ref = 1;
}
```

这时已经完成了pages的构建，然后还需要实现page_alloc和page_free来实现对page的分配和释放。

```
struct Page *
page_alloc(int alloc_flags)
{
    // Fill this function in
    struct Page* result = NULL;
    if(page_free_list){
        result = page_free_list;
        page_free_list = (*result).pp_link;
    }else{
        return NULL;
    }
    if(alloc_flags & ALLOC_ZERO){
        void *add = page2kva(result);
        memset(add,'\0',PGSIZE);
    }
    return result;
}
```

```
void
page_free(struct Page *pp)
{
    // Fill this function in
    (*pp).pp_link = page_free_list;
    page_free_list = pp;
}
```

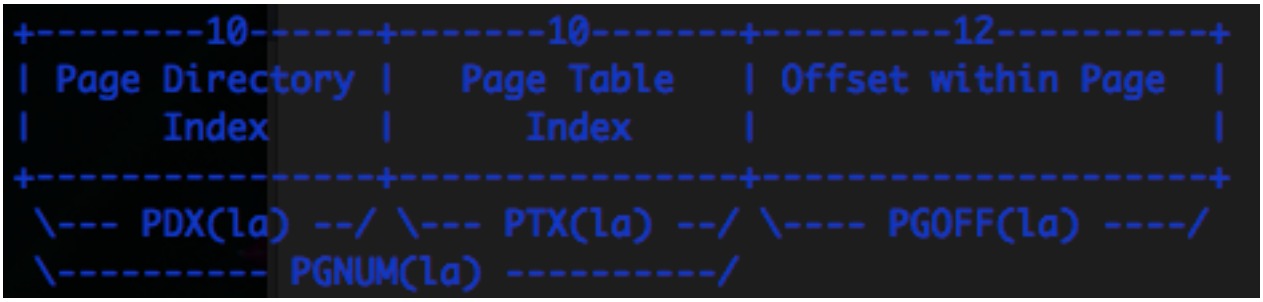
然后实现page_alloc_4pages和page_free_4pages，在alloc是通过连续向后取4个page，然后检查地址的差距来确定这4个page是不是连续的。

```
struct Page *
page_alloc_4pages(int alloc_flags)
{
    struct Page *head = page_free_list;
    struct Page *tail = head;
    struct Page *faketail = NULL;
    size_t i = 0;
    for(; i < 3; ++i)
        head = (*head).pp_link;
    while(head != NULL && head - tail != 3){
        head = (*head).pp_link;
        tail = (*tail).pp_link;
    }
    if(head == NULL)
        return NULL;
    faketail = tail;
    if(alloc_flags & ALLOC_ZERO){
        while(tail != head){
            void *add = page2kva(tail);
            memset(add,'\0',PGSIZE);
            tail = (*tail).pp_link;
        }
        void *add = page2kva(tail);
        memset(add,'\0',PGSIZE);
    }
    // Fill this function
    return faketail;
}
```

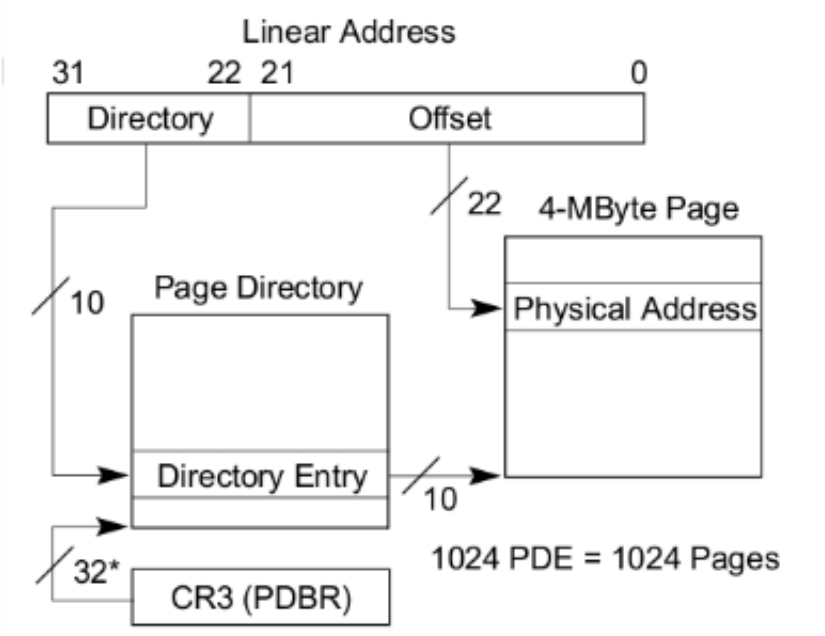
Part2. 虚拟地址内存

在实现虚拟内存时，必须先构建页表以实现地址的映射，由于有两级页表，第一级是page directory，第二级是page ta-

ble，分别对应虚拟地址的10bit长度，其结构可以如下图所示。



下图则表示了从一个virtual address通过二级页表翻译到实际物理内存的过程。



pdt*是指向page directory的指针，pgdir_walk作用就是把一个virtual address反映称对应的pt页表项，如果没有则根据要求创建。在pdt和和pte中存储的都是实际的物理地址。

boot_map_region则是映射一片虚拟地址到制定的物理地址。在实现的时候则是通过pgdir_walk创建对应页表项。

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    if(!pgdir)
        panic("pgdir is NULL");
    pde_t pde;
    pte_t *ptable;
    pde = pgdir[PDX(va)];
    if( pde & PTE_P ){
        ptable = (pte_t *)KADDR(PTE_ADDR(pde)); //pde is physical address of
            corresponding ptablej
        return &ptable[PTX(va)];
    }
    if(!create)
        return NULL;
    struct Page *newpage = page_alloc(ALLOC_ZERO);
    if(newpage == NULL)
        return NULL;
    ++newpage->pp_ref;
    pgdir[PDX(va)] = page2pa(newpage) | PTE_P | PTE_W | PTE_U;
    return &((pte_t*)page2kva(newpage))[PTX(va)];
}
```

page_lookup则实现了从virtual address查找对应的物理内存page，同样是通过pgdir_walk找到对应的pte,然后通过pa2page转换成对应page。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(!pte)
        return NULL;
    if(pte_store){
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

page_insert则是实现新建virtual address到physical address的映射。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    // Fill this function in
    physaddr_t add = page2pa(pp);
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(!pte){
        pte = pgdir_walk(pgdir, va, 1);
        if(!pte)
            return -E_NO_MEM;
    }else{
        if(*pte & PTE_P)
            page_remove(pgdir,va);
    }
    if(pp == page_free_list)
        page_free_list = page_free_list->pp_link;
    *pte = add | PTE_P | perm;
    pp->pp_ref++;
    return 0;
}
```

page_lookup则是查找一个virtual address对应的Page。

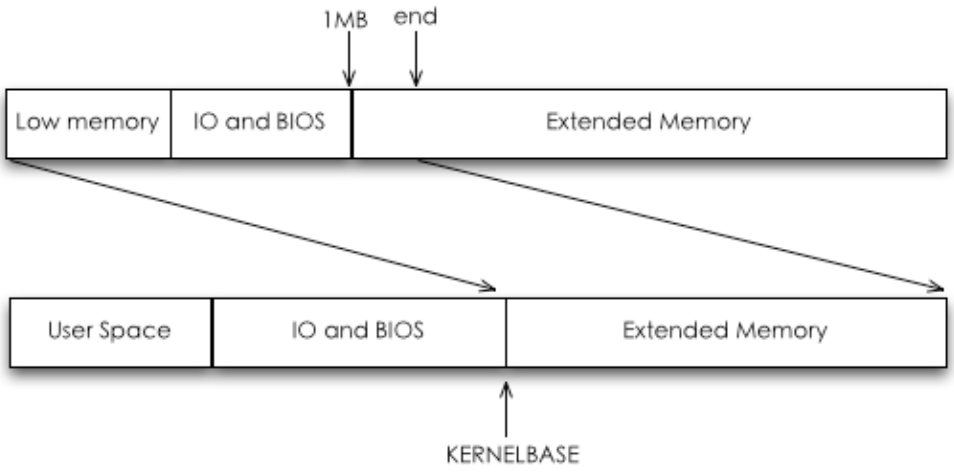
```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(!pte)
        return NULL;
    if(pte_store){
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

现在虚拟地址基本已经实现。

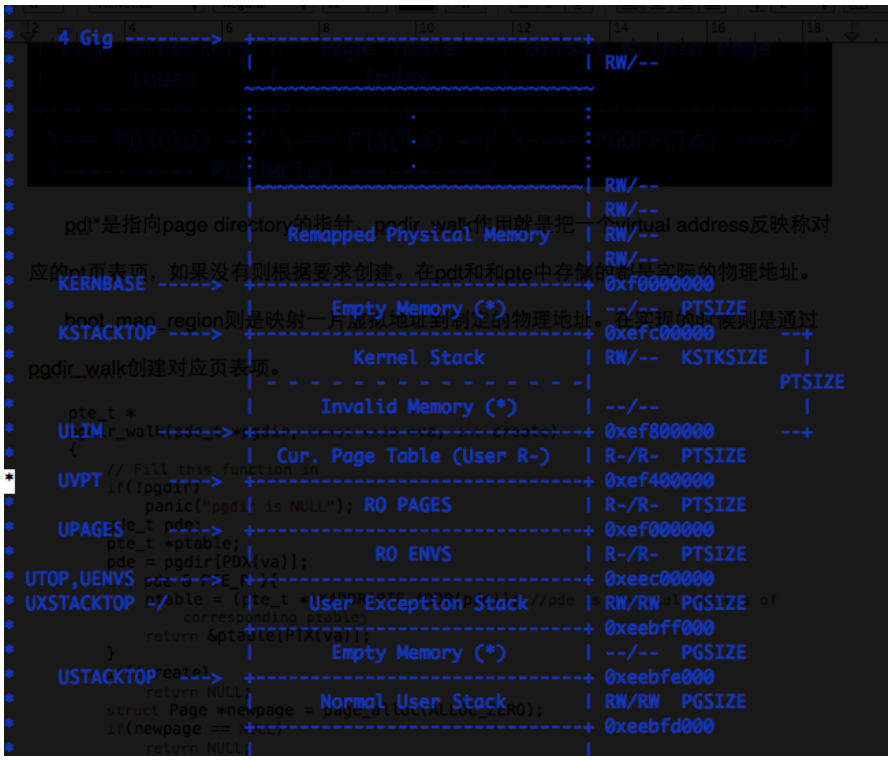
Part3. Kernel Address Space

在实现了基本的虚拟内存后，需要对内核的虚拟地址映射到实际load的物理地址，这就解决了link address和实际加载的内存地址不相符和的原因，之前entry.S中加载了entry_pgdir这个大页表，可以保证在virtual address完成前，内核空间的代码可以正常运行

下图展示了一部分映射的过程



而整个虚拟地址的分布则如下图所示




```
    boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages * sizeof(s-
truct Page), PGSIZE), PADDR(pages), PTE_U | PTE_P);
    boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PAD-
DR(bootstack), PTE_W);
    boot_map_region(kern_pgdir, KERNBASE, ROUNDUP(0xffffffff-
KERNBASE, PGSIZE), 0, PTE_W);
```

第一句话映射pages结构到UPAGES，第二句话映射了KernelStack，最后一句话则映射KernelBase到最后的所有内存。

Question 3:

Pde和pte有U/S和R/W位可以保护内核内存地址的访问，防止来自用户程序的访问。

Qeestion 4:

由于是32位，所以支持4G。

Question 5

page directory占用了4KB的空间，pages数组占用了4MB，page table占用了4MB

Question 6:

```
move $relocated, %eax
jmp %eax
```

这两句话把EIP转移到KERNELBASE以上，在此之前如在此报告上文中所述，从0开始的一个大页表（4MB）已经映射到了KERNELBASE开始的空间

```
 movl    $(RELOC(entry_pgdir)), %eax
 movl    %eax, %cr3
 # Turn on paging.
```