Lab2 作业报告

515030910489 丁贞锴

1 概述

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。
Lab2 相比 lab1 多了五个文件，memlayout.h pmap.c pmap.h kclock.h kclock.c 我们先来看看这五个文件讲了什么。
Memlayout.h: 这个头文件里面主要描述了虚拟内存的框架，定义了一些常量和 page 的数据结构。
pmap.c/pmap.h: 这是内存管理的主要文件，关于虚拟地址转换为物理地址，页表的建立，页的分配等相关函数都在这里面。
kclock.c/kclock.h:这个是更底层的实现,不过在本次 lab 里面,可以不考虑这两个文件。

2 Part 1: 物理页管理

这部分主要就是做一些初始化的工作和物理页的分配。
Exercise1 要求我们实现五个函数 boot_alloc() mem_init() page_init() page_alloc()

page_free()。
我们先看一下boot_alloc()函数讲了什么。

```
static void *
boot_alloc(uint32_t n)
{
        if(n<0){
           return NULL;
        }
    static char *nextfree;  // virtual address of next byte of free memory
        // static variable will not be deleted when the fuction is gone, it will
        // stay as time goes
    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.

        // end points the end of kernel in memory
    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 nextfree;
        result = nextfree; // start address
        nextfree += n;
        nextfree = KADDR(PADDR(nextfree));
        // means noting ,just nextfree = nextfree -KNERBASE + KNERBASE
        // just to check whether or not ,nextfree out of memory , just check if
panic

        nextfree = ROUNDUP(nextfree, PGSIZE);
        nextfree = KADDR(PADDR(nextfree));

    return result;
}
```

boot_alloc(n) 就是分配足够容纳 n 个 byte 的页，返回空闲页的起始地址。
注意 boot_alloc()函数里面有一个静态变量 nextfree，这个是个全局变量。
nextfree 指的是下一个没有被分配的页，注意这里面有一个 ROUNDUP（nextfree, PGSIZE）函数，这个函数的作用是把 nextfree 变成 PGSIZE 的倍数。
我们再看下一个函数 mem_init()。

```
void
mem_init(void)
{
    uint32_t cr0;
    size_t n;

    // Find out how much memory the machine has (npages & npages_basemem).
    i386_detect_memory();

    // Remove this line when you're ready to test this function.
    //panic("mem_init: This function is not finished\n");

    //////////////////////////////////////////////////////////////////////
    // create initial page directory.
    kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
    memset(kern_pgdir, 0, PGSIZE);

    //////////////////////////////////////////////////////////////////////
    // Recursively insert PD in itself as a page table, to form
    // a virtual page table at virtual address UVPT.
    // (For now, you don't have understand the greater purpose of the
    // following two lines.)

    // Permissions: kernel R, user R
    kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

    //////////////////////////////////////////////////////////////////////
    // Allocate an array of npages 'struct Page's and store it in 'pages'.
    // The kernel uses this array to keep track of physical pages: for
    // each physical page, there is a corresponding struct Page in this
    // array.  'npages' is the number of physical pages in memory.
    // Your code goes here:
        pages = boot_alloc(npages * sizeof(struct Page));
        memset(pages, 0, PGSIZE);


    //////////////////////////////////////////////////////////////////////
    // Now that we've allocated the initial kernel data structures, we set
    // up the list of free physical pages. Once we've done so, all further
```

```
    // memory management will go through the page_* functions. In
    // particular, we can now map memory using boot_map_region
    // or page_insert
    page_init();

    check_page_free_list(1);
    check_page_alloc();
    check_page();
}
```

这里就是初始化了页表,给页表分配了内存,也给 pages(里面存的是物理页的信息 pp_link, pp_ref) 分配了内存。

我们再看 page_init()。

```
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!
        pages[0].pp_ref = 1;
    pages[0].pp_link = NULL;
    size_t i;
    for (i = 1; i < npages_basemem; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    for (i = PGNUM(PADDR(boot_alloc(0))); i < npages; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
```

```
        page_free_list = &pages[i];
    }

    chunk_list = NULL;
}
```

page_init()函数就是把刚刚分配的页初始化一下，一共 npages 个页，但是要注意这中间有一部分页是 IO 的，不能碰，不过我们可以用 boot_alloc(0)来获取 nextfree。

我们再看 page_alloc()。

```
struct Page *
page_alloc(int alloc_flags)
{
    // Fill this function in
        if(!page_free_list) return NULL;
        struct Page * returnpage = page_free_list;
        page_free_list = page_free_list->pp_link;

        if(alloc_flags & ALLOC_ZERO){
           memset(page2kva(returnpage), 0, PGSIZE);
        }

    return returnpage;
}
```

page_alloc()函数就是取当前的 nextfree 并且返回，并且更新 nextfree。

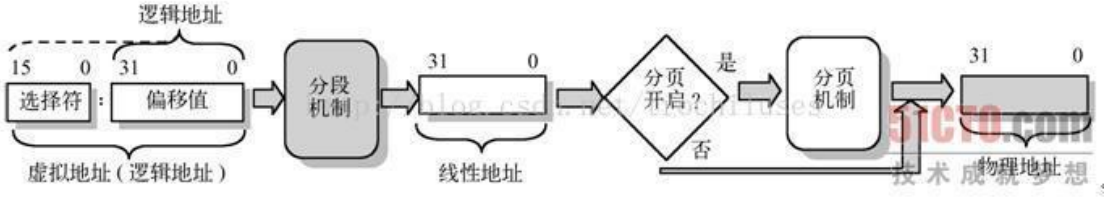
我们再看 page_free()。

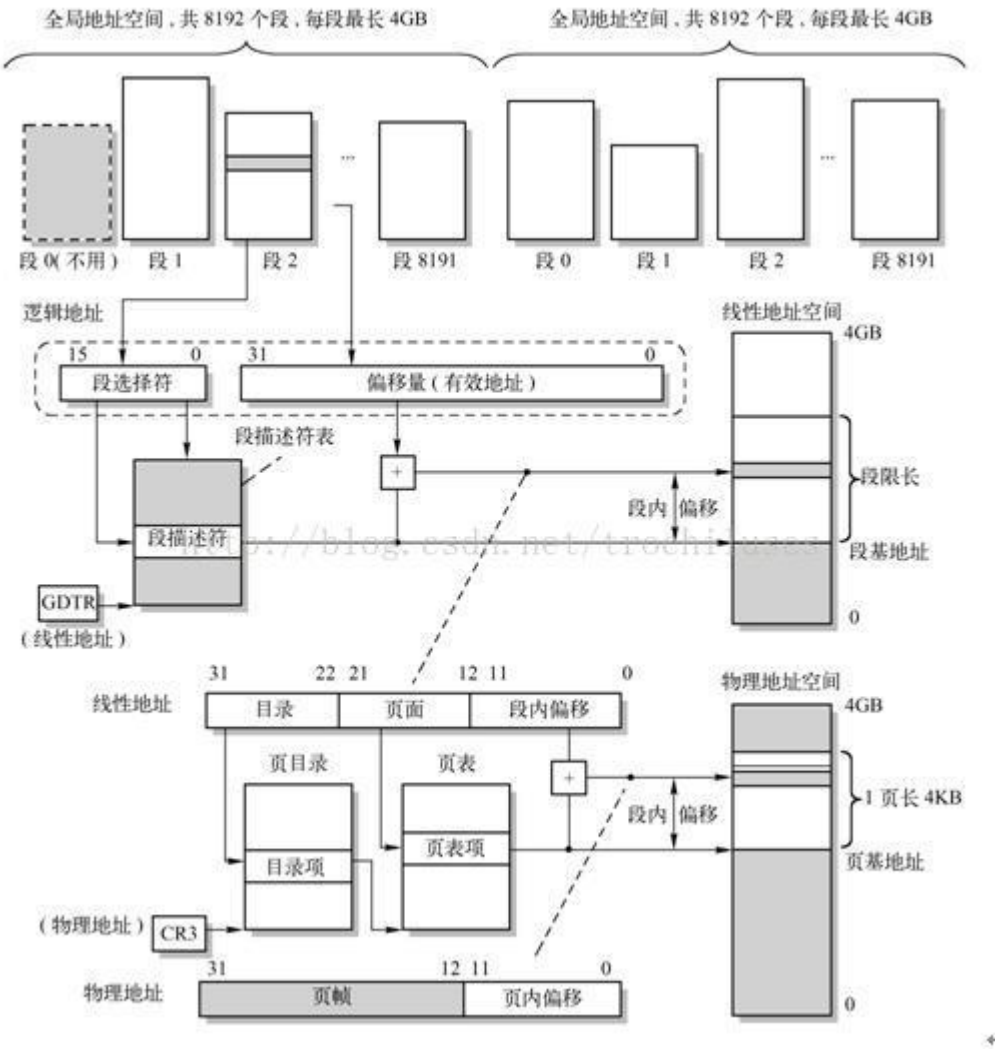
```
void
page_free(struct Page *pp)
{
    // Fill this function in
        if(pp->pp_ref==0){
           pp->pp_ref = 0;
       pp->pp_link = page_free_list;
       page_free_list = pp;
        }
}
```

page_free()就是把当前页的 pp_ref 置为 0，然后把当前页加到 nextfree 里面。Nextfree 其实是个链表结构。

3 Part 2 虚拟内存

这里首先讲述了虚拟地址，线性地址跟物理地址的关系，我找了下面两张图来更具像地解释这三者之间的关系。





在这部分，我们主要是实现虚拟地址跟物理地址之间的映射。我们一共要实现五个函数 pgdir_walk() boot_map_region() page_lookup() page_remove() page_insert(). 我们先看 pgdir_walk()。

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
        /*char *p1 = va>>22;
        char *PDE = pgdir + p1;
        char *p2 = (va<<10)>>22;
        char *PTE = (*PDE) + p2;*/
```

```
        pde_t *PDE = PDX(va) + pgdir;
        pte_t *PTE;
        if( (*PDE) & PTE_P){
            pte_t *PPN = (pte_t *)KADDR(PTE_ADDR(*PDE));
            PTE = PPN + PTX(va);
        }
        else if(create == 0) return NULL;
        else{
            struct Page *new_page = page_alloc(ALLOC_ZERO);
            if( new_page == NULL){
                return NULL;
            }
            new_page->pp_ref++;
            *PDE = page2pa(new_page) | PTE_P | PTE_W | PTE_U;
            pte_t *PPN = (pte_t *)KADDR(PTE_ADDR(*PDE));
            PTE = PPN + PTX(va);
        }
    return PTE;
}
```

page_walk()函数的作用就是把虚拟地址转换为物理地址。具体就是先根据虚拟地址的高 10 位在 pgdir 里面找到 page_table,然后再根据中间的 10 位在 page_table 里面找到对应的 PTE。

我们再看看 boot_map_region()。

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

这个函数就是把 pa 映射到 va 的位置，一共映射 size 个 byte。

我们再看看 page_lookup()。

```
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==NULL) || ((*PTE)&PTE_P) ==0) return NULL;
```

```
        if(pte_store!=0){
           *pte_store = PTE;
        }
        return pa2page(PTE_ADDR(*PTE));
}
```

这个函数就是根据 va 找到对应的物理页。

我们再看看 page_remove()

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

     pte_t **PTE = &pgdir ;
     struct Page *lookpage = page_lookup(pgdir, va, PTE);
     if(lookpage){
        page_decref(lookpage);
        **PTE = 0;
        tlb_invalidate(pgdir,va);
     }
}
```

这个函数就是把 va 对应的物理页 remove，其实就是 pp_ref 减一,并且当 pp_ref 小于 1 的时候，释放这个页。

我们再看看 page_insert()。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
     pte_t *PTE = pgdir_walk(pgdir, va, 1);
     if(PTE==NULL) return -E_NO_MEM;
     /*if((*PTE)&PTE_P){
         if(PTE_ADDR(*PTE) == page2pa(pp)){
            tlb_invalidate(pgdir,va);
            pp->pp_ref--;

         }
         else{
            page_remove(pgdir, va);
         }
         //page_remove(pgdir, va);
     }
     *PTE = page2pa(pp) | perm | PTE_P;
     pp->pp_ref++;
     return 0;*/
     if(((*PTE)&PTE_P) == 0){
              ;
     }
```

```
    else if(PTE_ADDR(*PTE) == page2pa(pp)){
        tlb_invalidate(pgdir,va);
        pp->pp_ref--;
    }
    else{
        page_remove(pgdir, va);
    }
    *PTE = page2pa(pp) | perm | PTE_P;
    pp->pp_ref++;
    return 0;

}
```

这个函数就是把物理页 pp 置为 va 对应的物理页。我一开始想的是，删掉之前 va 对应的物理页，然后把 pp 插进去。后来我看注释，要考虑之前 va 对应的物理页其实就是 pp 的情况，在这种情况下，要删除之前 TLB 里面的条目。

4 Part 3 内核地址空间

这部分我们要开始做内核地址空间的映射，把相应的物理页映射到对应的虚拟地址空间。我们看最开始的函数 mem_init()

```
void
mem_init(void)
{
    //////////////////////////////////////////////////////////////////////
    // Now we set up virtual memory

    //////////////////////////////////////////////////////////////////////
    // Map 'pages' read-only by the user at linear address UPAGES
    // Permissions:
    //    - the new image at UPAGES -- kernel R, user R
    //      (ie. perm = PTE_U | PTE_P)
    //    - pages itself -- kernel RW, user NONE
    // Your code goes here:
        boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);

    //////////////////////////////////////////////////////////////////////
    // Use the physical memory that 'bootstack' refers to as the kernel
    // stack.  The kernel stack grows down from virtual address KSTACKTOP.
    // We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
    // to be the kernel stack, but break this into two pieces:
    //     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
    //     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
    //       the kernel overflows its stack, it will fault rather than
    //       overwrite memory.  Known as a "guard page".
```

```
    //     Permissions: kernel RW, user NONE
    // Your code goes here:
        boot_map_region(kern_pgdir,        KSTACKTOP-KSTKSIZE,        KSTKSIZE,
PADDR(bootstack), PTE_W);

    ////////////////////////////////////////////////////////////////////
    // Map all of physical memory at KERNBASE.
    // Ie.  the VA range [KERNBASE, 2^32) should map to
    //      the PA range [0, 2^32 - KERNBASE)
    // We might not have 2^32 - KERNBASE bytes of physical memory, but
    // we just set up the mapping anyway.
    // Permissions: kernel RW, user NONE
    // Your code goes here:
        lcr4(rcr4() | CR4_PSE);
        boot_map_region_large(kern_pgdir, KERNBASE, 0x10000000 , 0, PTE_W);
        //boot_map_region(kern_pgdir, KERNBASE, 0x10000000 , 0, PTE_W);

    // Check that the initial page directory has been set up correctly.
    check_kern_pgdir();

    // Switch from the minimal entry page directory to the full kern_pgdir
    // page table we just created.  Our instruction pointer should be
    // somewhere between KERNBASE and KERNBASE+4MB right now, which is
    // mapped the same way by both page tables.
    //
    // If the machine reboots at this point, you've probably set up your
    // kern_pgdir wrong.
    lcr3(PADDR(kern_pgdir));

    check_page_free_list(0);

    // entry.S set the really important flags in cr0 (including enabling
    // paging).  Here we configure the rest of the flags that we care about.
    cr0 = rcr0();
    cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_MP;
    cr0 &= ~(CR0_TS|CR0_EM);
    lcr0(cr0);

    // Some more checks, only possible after kern_pgdir is installed.
    check_page_installed_pgdir();
}
```

我们要加的其实就是三个 boot_map_region。boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);映射页表。boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W); 映射内核栈。

boot_map_region_large(kern_pgdir, KERNBASE, 0x10000000 , 0, PTE_W);映射 256MB 的物理内存。

5 总结

这个 lab 主要实现内存管理相关的工作，具体就是实现虚拟内存的 layout，物理内存的 layout，已经虚拟内存与物理内存之间的映射。我在这里面也踩了很多坑，比如对物理内存里面的各个部分里面存了什么理解有误，尤其是 pages 那一块，后来反复看代码才知道了。地址间的映射要注意地址有没有越界,在页表那里,我对于也表里面存的是什么产生了误解。我一开始以为二级页表里面存的 PTE 就直接是加了 offset 的物理地址,其实没有,只是 PPN。