

JOS实习报告2

林海南, 00948704,
lhainan09@163.com

March 25, 2012

Contents

1 TASK	2
2 Exercise	2
2.1 Part 1: Physical Page Management	2
2.2 Part 2: Virtual Memory	6
2.2.1 Virtual, Linear, and Physical Addresses	6
2.2.2 Reference counting	8
2.2.3 Page Table Management	8
2.3 Part 3: Kernel Address Space	13
2.3.1 Permissions and Fault Isolation	13
2.3.2 Initializing the Kernel Address Space	15
3 RESULT	24
4 PROBLEM AND SOLUTION	25
5 THINKING AND HAVEST	26
6 Suggestions	26
7 Referance	26

1 TASK

第一周, 调研并完成了lab1, 和lab2;

第二周, 调研并完成了lab3 lab4 lab5;

第三周, 完成了报告了challenge1 和challenge2

2 Exercise

2.1 Part 1: Physical Page Management

Exercise 1. In the file kern/pmap.c, you must implement code for the following functions (probably in the order given).

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

check_page_free_list() and check_page_alloc() test your physical page allocator. You should boot JOS and see whether check_page_alloc() reports success. Fix your code so that it passes. You may find it helpful to add your own assert()s to verify that your assumptions are correct.

boot_alloc()是一个直接分配物理页面功能的函数, 它仅仅在建立虚拟地址映射时被使用。它的基本原理是根据设定的可用地址指针 (实际上是kernel以上的地址被用来使用) 来判断那些物理地址可以被使用, 并且分配时要按找页的大小进行对其对齐, 这也是为了建立虚拟页式映射做准备。

对于boot_alloc()的功能提示, 有如下要求:

- 1.如果n大于0, 那么申请一个空间足够容纳n字节, 并且大小要与页面大小对齐, 并返回这块空间的起始地址 (虚拟地址)
- 2.如果n=0, 那么直接返回当前可用地址的起始地址 (虚拟地址)
- 3.要考虑n超出系统所能容纳的上限。

这里, 我们首先观察boot_alloc()中已有的定义, 我们会发现, 其中的end[]指针指向的是kernel的底端, 是可以利用的内存位置开始的地址, 而nextfree指针的定义则是用来规定可用的位置, 它是与页大小对齐过得, 可以直接应用, 而我们在内存管理申请物理内存也是从它开始进行申请的, 因此, 若要申请某个大小的空间, 需要做的就是改变nextfree指向的位置, 但是要保证nextfree与页面大小对齐。

关于申请内存上限, 也是看了网上的一些讨论后受到了启发。在上一次开启页式的时候, 设定了物理的4MB映射到kernbase+4MB的虚拟地址, 因此对应的超出这个区域范围的地址范围其实应该无法访问到, 因此这里我们要将内存申请的范围定在kernbase+4MB内。

```

1      // Allocate a chunk large enough to hold 'n' bytes, then update
2      // nextfree. Make sure nextfree is kept aligned
3      // to a multiple of PGSIZE.
4      //
5      // LAB 2: Your code here.
6      if((physaddr_t)nextfree+n>KERNBASE+NPTENTRIES*PGSIZE) //if the
          application out of memory, panic
7          panic("out_of_memory!\n");
8      if(n>0)
9      {
10         result = nextfree;                //record
            current available VA
11         nextfree = (char *)((uint32_t)nextfree + n); //update
            the nextfree
12         nextfree = ROUNDUP((char *)nextfree,PGSIZE); //align
            the pointer
13         return result;                    //return
            starting VA of the allocation
14     }
15     else if(n==0)
16         return nextfree;                //return
            current available VA

```

page_init()实现的功能是，初始化所有的页，使得可用的页面都加入空闲页链表，方便之后使用，同时，没有在这些链表中的页面即为正在使用或者不可用。

这里我们首先查看Page结构体：

```

1 struct Page {
2     // Next page on the free list.
3     struct Page *pp_link;
4
5     // pp_ref is the count of pointers (usually in page table
        entries)
6     // to this page, for pages allocated using page_alloc.
7     // Pages allocated at boot time using pmap.c's
8     // boot_alloc do not have valid reference count fields.
9
10    uint16_t pp_ref;
11 };

```

其中，pp_link用来构成空闲页链表，为指向下一个空闲页的指针。pp_ref表示对这个页面引用的次数；并且，可用页面在物理空间的分布如下：

- 1 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 do not, but...)
- 2 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE) is free.
- 3 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must never be allocated.
- 4 4) Then extended memory [EXTPHYSMEM, ...).
Some of it is in use, some is free. Where is the kernel

10 in physical memory? Which pages are already in use for
11 page tables and other data structures?

然后需要做的就是将可用页面加入空闲页链表。实现代码如下：

```

1 void
2 page_init(void)
3 {
4     uint32_t count;
5     count=0;
6     size_t i;
7     for (i = 0; i < npages; i++) {
8         pages[i].pp_ref = 0;
9         if(i==0)           //for page0
10             continue;
11         //for [PGSIZE, npages_basemem * PGSIZE)
12         if(i<npages_basemem && i>=1)
13         {
14             pages[i].pp_link = page_free_list;
15             page_free_list = &pages[i];
16             count++;
17         }
18         //for [IOPHYSMEM, EXTPHYSMEM)
19         if(i>=IOPHYSMEM/PGSIZE && i<EXTPHYSMEM/PGSIZE)
20             continue;
21
22         //for [EXTPHYSMEM, ...).
23         if(i>=EXTPHYSMEM/PGSIZE && i<PADDR(boot_alloc(0))/
24             PGSIZE)
25             continue;
26         if(i>=PADDR(boot_alloc(0))/PGSIZE)
27         {
28             pages[i].pp_link = page_free_list;
29             page_free_list = &pages[i];
30             count++;
31         }
32     }
33 }
```

page_alloc()完成的功能是申请一个物理页面，并且根据其中的参数判断是否将此页面0初始化。注意：这个操作并不改变对页面的引用。这里比较特殊的一处处理是对页面进行0初始化时，需要访问的指针应该是虚拟地址，因为我们是没办法对物理地址进行直接访问的。这个后边还会被提及，这里将Page结构体转换为相应的虚拟地址的函数是page2kva();因此，实现的代码如下：

```

1 struct Page *
2 page_alloc(int alloc_flags)
3 {
4     if(page_free_list == NULL)
5         return NULL;
6     struct Page * tmp_page;
7     // Fill this function in
8     //get a free page
9     tmp_page = page_free_list;
```

```

10     if (alloc_flags & ALLOC_ZERO)
11     {
12         //memset the page all zero
13         memset(page2kva(page_free_list), 0, PGSIZE);
14     }
15     //current page is removed from page free list
16     page_free_list = page_free_list->pp_link;
17     return tmp_page;
18 }
19

```

page_free()完成的功能是将一个页面释放到空闲页链表中，但是前提是没有任何的程序对这个页面有引用。代码如下：

```

1 void
2 page_free(struct Page *pp)
3 {
4     // Fill this function in
5     //do nothing
6     if (pp->pp_ref > 1)
7     {
8         return;
9     }
10    //add to page free list
11    pp->pp_link = page_free_list;
12    pp->pp_ref=0;
13    page_free_list = pp;
14    return ;
15 }

```

最后，我们了解下mem_init()，添加的语句比较简单我就不说了：

```

1 void
2 mem_init(void)
3 {
4     uint32_t cr0;
5     size_t n;
6
7     // Find out how much memory the machine has (npages &
8     // npages_basemem).
9     i386_detect_memory();
10
11    // Remove this line when you're ready to test this function.
12    // panic("mem_init: This function is not finished\n");
13
14    //
15    // //////////////////////////////////////
16
17    // create initial page directory.
18    kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
19    memset(kern_pgdir, 0, PGSIZE);
20
21    kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
22
23    // Your code goes here:

```

```
21     pages = (struct Page *)boot_alloc(sizeof(struct Page)*npages);
22     // or page_insert
23     page_init();
24     check_page_free_list(1);
25
```

这里, `mem_init()`完成的是, 申请了一块页目录, 并且设置了其中一个页目录项(用户和内核都是只读); 初始化空闲页链表, 然后对上述功能做检查。关于页表项功能位的设定下边还将有介绍, 这里不做过多说明。

2.2 Part 2: Virtual Memory

2.2.1 Virtual, Linear, and Physical Addresses

Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

首先, 我们应该先了解虚拟地址向物理地址转换的过程, 参看figure 1.

首先, 虚拟地址根据段基址和段选择符得到段描述符, 同地址中的便宜量相加得到线性地址。然后, 再将线性地址通过页目录和页表转换为物理地址。过程就不详细进行描述了。

然后在对页表项进行介绍, 参看Figure 2.

P 在不在位, 为0则说明页表项无效;

R/W 读写位, 1可写, 0只读

U/S 决定在管太下使用还是目态下使用

D 脏位, 页面是否做过修改

AVAIL 是否可被系统程序员使用

页式的内存保护主要是利用上述的U/S位、R/W位, 并且要将两级的页表保护位结合起来。

Figure 1: virtual address to physics address

Figure 5-12. 80386 Addressing Mechanism

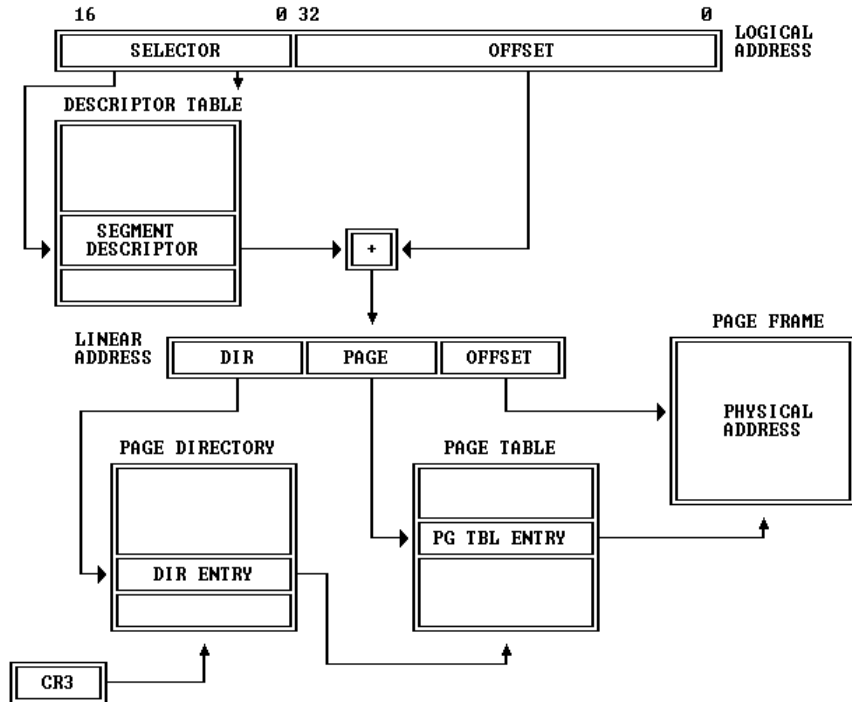
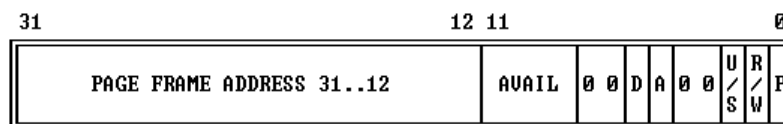


Figure 2: virtual address to physics address

Figure 5-10. Format of a Page Table Entry



P - PRESENT
 R/W - READ/WRITE
 U/S - USER/SUPERVISOR
 D - DIRTY
 AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE
 NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

Exercise 3. While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press `Ctrl-a c` in the terminal (the same binding returns to the serial console).

Use the `xp` command in the QEMU monitor and the `x` command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges of virtual memory are mapped and with what permissions.

```
(qemu) xp/12x 0x00100000
0000000000100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0000000000100010: 0x34000004 0x6000b812 0x220f0011 0xc0200fd8
0000000000100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0

(gdb) x/12x 0xf0100000
0xf0100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0xf0100010: 0x34000004 0x6000b812 0x220f0011 0xc0200fd8
0xf0100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
```

从上述结果可以看出，物理地址与对应的虚拟地址的内容是一样的。

Question

1. Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

这里是对虚拟地址进行操作的，因此应该用 `uintptr_t`。

2.2.2 Reference counting

对页面应用次数的记录一定要精确，否则会出现不该释放的页面被释放，或者应该释放的却一直无法释放。

实际上，对于UTOP以边的页面，在boot的时候就会被内核设定好，之后也不会被释放，因此，没必要对其进行引用的计数。还有注意，`boot.alloc()`中并没有对页面的引用计数进行修改。

2.2.3 Page Table Management

Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.

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

check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

这个exercise中，我们将要完成一系列与页表管理有关的函数。

这里，有几个必要的宏和函数。其中

page2pa() 计算页面对应的物理地址

pa2page() 计算物理地址对应的页面

PAADDR() 将虚拟地址转换为物理地址

VADDR() 将物理地址转换为虚拟地址

这里，我在介绍下怎样在代码中查找到对应VA的PA：

构造虚拟地址vaddr = UVPT[31:22]—PDX—PTX—00在虚拟内存空间里查询。根据两级页表翻译机制：

1. 系统首先取出vaddr的前10位，即UVPT[31:22]，去页目录里查询，根据pgdir[PDX(UVPT)] = PADDR(pgdir)—PTE_U—PTE_P；注意UVPT[31:22]等价于PDX(UVPT)，所以得到的二级页表地址A0，还是页目录pgdir本身。
2. 再取出vaddr的中间10位，即PDX，去二级页表中A0（即页目录），查找到的就是addr所在二级页表的物理地址A1，注意！！不是addr的物理页面地址，是二级页表的地址！！
3. 最后取出vaddr的最后12位，即PTX—00，去A1物理页中（即addr所在二级页表）查找，得到的就是addr最后所在页面的物理地址查找addr对应的二级页表物理地址：

构造虚拟地址vaddr = UVPT[31:22]—UVPT[31:22]—PDX—00在虚拟内存空间里查询。注意这个地址等价于PDX(UVPT)—PDX(UVPT)—PDX—00。

1. 根据上面的分析，我们可以知道页式转换的前两步，地址转换系统都会跳回到页目录本身

2. 最后一步取出vaddr的最后12位，即PDX—00，去pgdir的物理页查询，查到的就是addr对应的二级页表物理地址

pgdir_walk()用来得到一个线性地址对应的页目录项的地址。如果没有当前的页表，且传入的参数为create，那么就申请页面并创建一个页表。

```
1 pte_t *
2 pgdir_walk(pte_t *pgdir, const void *va, int create)
3 {
4     // Fill this function in
5     pte_t *tmp_pde;
6     tmp_pde = (pte_t *) (pgdir + PDX(va));
7     struct Page *tmp_page;
```

```

8      //if not exist , alloc a new page
9      if(((tmp_pde) & PTE_P)==0){
10         if(create==0)
11             return NULL;
12         else{
13             tmp_page=page_alloc(1);
14             //if page can't be allocated,return 0
15             if(!tmp_page)
16                 return NULL;
17             else //update page directory entry and return
                  //the available addr
18                 {
19                     tmp_page->pp_ref++;
20                     *tmp_pde &=0xfff;
21                     *tmp_pde |=PTE_U|PTE_P|PTE_W;
22                     *tmp_pde |=page2pa(tmp_page);
23                     return (pte_t *)KADDR((physaddr_t)((
                        pte_t *)page2pa(tmp_page)+PTX(va)))
                        ;
24                 }
25             }
26         }
27     else{
28         return (pte_t *)KADDR((physaddr_t)((pte_t *)PTE_ADDR(*
                        tmp_pde)+PTX(va)));
29     }
30 }

```

这里，在pde表项的更新上，与同学讨论后，最后觉得这样处理较好。考虑为什么要将*tmp_pde |=PTE_U|PTE_P|PTE.W;

这里主要是为了看他所给的提示2。考虑x86系统的安全保护判断过程：

1. 先检查段权限位DPL，这个是所访问数据段或者Call gate的权限级别，和当前权限级别CPL进行比较，如果不够则产生权限异常（具体机制请参考手册，在这个问题上我们不用管它），否则进入下一步
2. 再检查页目录相应表项的访问权限，如果不够也产生异常
3. 最后检查二级页表相应表项的访问权限，不够就产生异常

这里，我们查看IA32手册中页表和页目录的组合控制结果，见Figure 3.

这里，我们看到，如果页目录项设置为supervisor方式，那么无论如何他对整个空间地址都是可读可写的，因此，我们要将PTE_U设置为user模式。

因此，在页目录项的保护位要设置的权限更开放些，将权限的控制交到页表项来控制，这样，就可以避免了很多复杂的处理，相对来讲比较简单。

boot_map_region()将虚拟地址[va,va+size]与物理地址[pa,pa+size]之间建立了映射。其本质就是更新对应的页目录和页表。

Figure 3: combined PDE and PTE Protection

Table 4-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

```

1 static void
2 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
3                 int perm)
4 {
5     // Fill this function in
6     pte_t *tmp_pte;
7     int i;
8     pde_t *tmp_pde;
9     for (i=0; i<size/PGSIZE; i++)
10     {
11         //update pde
12         tmp_pde=(pde_t *) (pgdir+PDX(va));
13         tmp_pte=pgdir_walk(pgdir, (void *)va,1);
14
15         //update pte
16         *tmp_pte =0;
17         *tmp_pte |=perm|PTE_P;
18         *tmp_pte |= (physaddr_t) (pa+PGSIZE*i);
19         va+=PGSIZE;
20     }
21 }

```

对于page_insert(),设置页表也页目录,使得页page与虚拟地址va建立一个映射。

其实现有几个细节,首先,我们当前的va可能已经有了一个页面与之对应,那么我们应该先将其removed;其次,允许页表不存在时申请页面并更新页

目录；最后，此处建立页面对虚拟地址的映射，其实完成了对页面的一次引用，应该使每个建立映射的页面的引用加1。

这里可能还会遇到一个极端情况，就是可能有同一页面重复的与相同的va建立映射，这时我的解决方法是，在每次在page_remove()前，对引用值进行加一，那么，当这种情况发生时，就不会发生将此页面误加入空闲页链表。

```

1 int
2 page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
3 {
4     // Fill this function in
5     pte_t *tmp_pte;
6     pde_t *tmp_pde;
7     tmp_pde = (pde_t *) (pgdir + PDX(va));
8     //find pte' address
9     tmp_pte = pgdir_walk(pgdir, va, 1);
10    //can't get such pte, return out of memory
11    if (tmp_pte == NULL)
12        return -E_NO_MEM;
13    //update pde and pte;
14    *tmp_pde |= PTE_P | perm;
15    pp->pp_ref++; //ref increased
16    //if rematched, remove it
17    page_remove(pgdir, va);
18    //update pte
19    *tmp_pte = 0;
20    *tmp_pte |= PTE_P | perm;
21    *tmp_pte |= page2pa(pp);
22    return 0;
23 }

```

page_lookup(), 根据虚拟地址找到一个页面（结构体）。

```

1 struct Page *
2 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
3 {
4     // Fill this function in
5     pte_t *tmp_pte;
6     if ((tmp_pte = (pte_t *) pgdir_walk(pgdir, va, 0)) == NULL)
7         return NULL;
8     else
9     {
10        if ((*tmp_pte & PTE_P) == 0)
11            return NULL;
12        if (pte_store != NULL)
13            *pte_store = tmp_pte;
14        return pa2page(PTE_ADDR(*tmp_pte));
15    }
16    return NULL;
17 }

```

page_remove(), 删除已有的va和page间的映射；实现要点为，页面引用要减少，如果降低到0了还要释放页面；若PTE存在，可能还涉及到PTE清零，和是TLB失效的操作。

```

1 void
2 page_remove(pde_t *pgdir, void *va)
3 {
4     // Fill this function in
5     struct Page *tmp_page;
6     pte_t *tmp_pte;
7     tmp_page=page_lookup(pgdir, va, &tmp_pte);
8     if(tmp_page==NULL)
9         return;
10    page_decref(tmp_page);
11    pde_t *tmp_pde;
12    pte_t *pte;
13    tmp_pde=(pde_t *) (pgdir+PDX(va));
14    if(*tmp_pde & PTE_P)
15    {
16        tmp_pte =(pte_t *) KADDR(PTE_ADDR(*tmp_pde));
17        pte=(pte_t *) (tmp_pte+PTX(va));
18        if(*pte & PTE_P)
19            tlb_invalidate(pgdir, va);
20        *pte=0;
21    }
22 }
23

```

将应该去掉的注释去掉，重新运行make qemu，就会发现page_check()和check_boot_pgdir()已经通过了，但是check_page_free_list(0)未通过。

2.3 Part 3: Kernel Address Space

2.3.1 Permissions and Fault Isolation

这里，我们观察下inc/memlayout.h中的空间布局。

```

1  /*
2  * Virtual memory map:
3  *
4  *
5  * 4 Gig -----> +-----+
6  * |               | RW/--
7  * |               |
8  * |               |
9  * |               |
10 * |               |
11 * |               | RW/--
12 * |               | RW/--
13 * | Remapped Physical Memory | RW/--
14 * |               | RW/--
15 * KERNBASE -----> +-----+ 0xf0000000
16 * | Empty Memory (*) | --/-- PTSIZE
17 * KSTACKTOP -----> +-----+ 0xefc00000
18 *   --+
19 * |               | Kernel Stack | RW/-- KSTKSIZE
20 *

```

```

19 *          | - - - - - |
20 *          |          PTSIZE          |
21 *          | Invalid Memory (*)      | --/--
22 *          |          |
23 *          |          |
24 *          |          |
25 *          |          |
26 *          |          |
27 *          |          |
28 *          |          |
29 *          |          |
30 *          |          |
31 *          |          |
32 *          |          |
33 *          |          |
34 *          |          |
35 *          |          |
36 *          |          |
37 *          |          |
38 *          |          |
39 *          |          |
40 *          |          |
41 *          |          |
42 *          |          |
43 *          |          |
44 *          |          |
45 *          |          |
46 *          |          |
47 *          |          |
48 *          |          |
49 *          |          |
50 *          |          |
51 *          |          |

```

[UTEXT,USTACKTOP)：权限kernel/user为RW/-
用户程序的.text段以及用户堆栈共用区域，两个区域从两头向中间生长。，并且用户堆栈部分并没有占据所有的整个这块区域，还有一部分未被开发利用。

[UPAGES,UVPT)：权限为R-/R-

在虚拟地址中这段内存就是对应的在实际物理地址里pages数组对应储存位置。（在后面的代码部分我们可以看到相应的操作语句）。可以看到这段地址在ULIM之下，也就是说操作系统开放pages数组便于让用户程序可以访问。为什么呢？比如说假如有的程序想要知道一个物理页面被引用了多少次，那

么根据相应的pages[i].pp ref就可以知道了。我们看到这个区域在布局中分配了PTSIZE的大小，那么这个大小够么？因为我们知道在物理页面中pages所占用的大小为 $\text{npage} \times \text{sizeof}(\text{struct Page}) = 12\text{B} \times \text{npage}$ 。我们看看PTSIZE的空间可以装struct Page的个数为 $1024 \times 4\text{KB} / 8\text{B} = 4\text{MB} / 8\text{B} = 0.5\text{M}$ 。一个Page结构对应一个实际大小为4KB的物理页面，所以这个PTSIZE大小的虚拟地址空间能够管理 $0.5\text{M} \times 4\text{KB} = 2.0\text{GB}$ 的物理内存，已经足够用来保存和管理本实验中的物理内存。

[UVPT,ULIM): 权限为R-/R-

这个地址映射到页目录所在的物理地址处，即原来在程序中看到的变量pgdir，开放给用户和系统都只读，可以使用户得到当前内存中某个虚拟地址对应的物理页面地址是多少。这段空间在虚拟地址上分配了PTSIZE，但是实际上只使用了物理页面上的PGSIZE个空间（可以回头去看看pgdir的空间分配参数是多少），所以这里很容易被误导和搞迷糊，请注意。这里我们查看下建立映射的代码：

```

1      //
2      // create initial page directory.
3      kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
4      memset(kern_pgdir, 0, PGSIZE);
5
6      //
7      // Recursively insert PD in itself as a page table, to form
8      // a virtual page table at virtual address UVPT.
9      // (For now, you don't have understand the greater purpose of
10     // the
11     // following two lines.)
12
13     // Permissions: kernel R, user R
14     kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

[KERNBASE,2³²): RW/-

这个部分映射实际物理内存中从0开始的中断向量表IDT、BIOS程序、IO端口以及操作系统内核等。即内核使用的虚拟地址KERNBASE + x进入这段地址查找到实际上的物理地址就是它自身。所以这段空间是为操作系统准备的。

2.3.2 Initializing the Kernel Address Space

Exercise 5. Fill in the missing code in mem_init() after the call to check_page(). Your code should now pass the check_kern_pgdir() and check_page_installed_pgdir() checks.

这里，我主要完成了一系列的物理地址到虚拟地址的映射，并设置了相应的权限。然后再次重新运行make qemu，发现整个lab都已经通过了。

```

lhn@ubuntu:~/6.828/lab1$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
qemu -hda obj/kern/kernel.img -serial mon:stdio
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
f011bfe8
0
0
0
0
check_page_alloc() succeeded!
freelist 0
current breakpoint
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

在建立映射时，我遇到了一个问题，在代码中，给出的提示如下：

```

1 //
2 // Map 'pages' read-only by the user at linear address UPAGES
3 // Permissions:
4 //   - the new image at UPAGES -- kernel R, user R
5 //   (ie. perm = PTE_U | PTE_P)
6 //   - pages itself -- kernel RW, user NONE
7 // Your code goes here:
8 boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_P |
   PTE_U);

```

这里，对于UPAGES段的权限问题，让我觉得很苦恼，他说page itself是内核可读写的，但是本身这段地址是不可写的，仔细理解后，发现，其实原理是这个样子的：对于page，我们已经有了一个虚拟地址对其进行访问，他是在kernbase之上的某个位置，而现在需要建立的位置是KERNBASE之下的，因此这里只是要设定用户访问这个位置的地址来访问pages的内容时是可读的，而内核通过之前的地址访问pages就是可读写的。这样理解就通顺了。

Question

2. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

3. (From Lecture 3) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

4. What is the maximum amount of physical memory that this operating system can support? Why?

5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

6. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

2. 映射表格如下

Entry	Base Virtual Address	Points to (logically)
1023	0xffc00000	Page table for top 4MB of phys memory
.	?	?
985	0xef800000	Kernel Stack and some invalid memory
984	0xef400000	current page table
983	0xef000000	Pages arrays
.	?	?
.	?	?
.	?	?
0	0x00000000	[see next question]

还有一些没有用到过得匹配:

Entry	Base Virtual Address	Points to (logically)
981	0xeebfff00	User Exception Stack
.	?	?
979	0xeebfd000	Normal User Stack

3. 其实上边的报告已经对此问题做了回答。首先，程序访问内存实际上通过虚拟内存来进行实现的，而虚拟地址向物理地址的转换则是通过两级页表来实现的，在页表项和页目录项中，我们已经设置了相应的保护位，通过这两个保护位，我们保证了用户不能访问内核部分的内存。

4. 在虚拟地址中这段内存就是对应的在实际物理地址里pages数组对应储存位置。（在后面的代码部分我们可以看到相应的操作语句）。可以看到这段地址在ULIM之下，也就是说操作系统开放pages数组便于让用户程序可以访问。为什么呢？比如说假如有的程序想要知道一个物理页面被引用了多少次，那么根据相应的pages[i].pp ref就可以知道了。我们看到这个区域在布局中分配了PTSIZE的大小，那么这个大小够么？因为我们知道在物理页面中pages所占用的大小为 $npage \times \text{sizeof}(\text{struct Page}) = 12B \times npage$ 。我们看看PTSIZE的空间可以装struct Page的个数为 $1024 \times 4KB / 8B = 4MB / 8B = 0.5M$ 。一个Page结构对应一个实际大小为4KB的物理页面，所以这个PTSIZE大小的虚拟地址空间能够管理 $0.5M \times 4KB = 2GB$ 的物理内存，已经足够用来保存和管理本实验中的

物理内存。

5.首先, 分析管理内存需要的额外开销都有什么: 页目录, 页表, Pages数组; 共需4MB+4MB+4KB大小的开销。

6.利用gdb工具查看kernel运行的状况, 运行到如下代码

```

1 f0100020: 0d 01 00 01 80      or    $0x80010001,%eax
2      movl    %eax, %cr0
3 f0100025: 0f 22 c0            mov    %eax,%cr0
4
5      # Now paging is enabled, but we're still running at a low EIP
6      # (why is this okay?).  Jump up above KERNBASE before entering
7      # C code.
8      mov    $relocated, %eax
9 f0100028: b8 2f 00 10 f0      mov    $0xf010002f,%eax
10     jmp     *%eax
11 f010002d: ff e0              jmp     *%eax

```

利用gdb可以发现, 在执行到f010002d之前, EIP一直处在低地址段, 执行了跳转指令后, 才到了高地址段开始执行, 明显, 在开启页式之后并没有立刻的进行地址的重定位, 只有在上述指令执行过后, 才开始。

关于在kernel.asm中前述地址为什么都是0xf0100000段的, 我认为是因为反汇编时是以VMA 最为标准的, 在gdb调试的时候则不是。而objdump看到的VMA也正好就是0xf0100000。

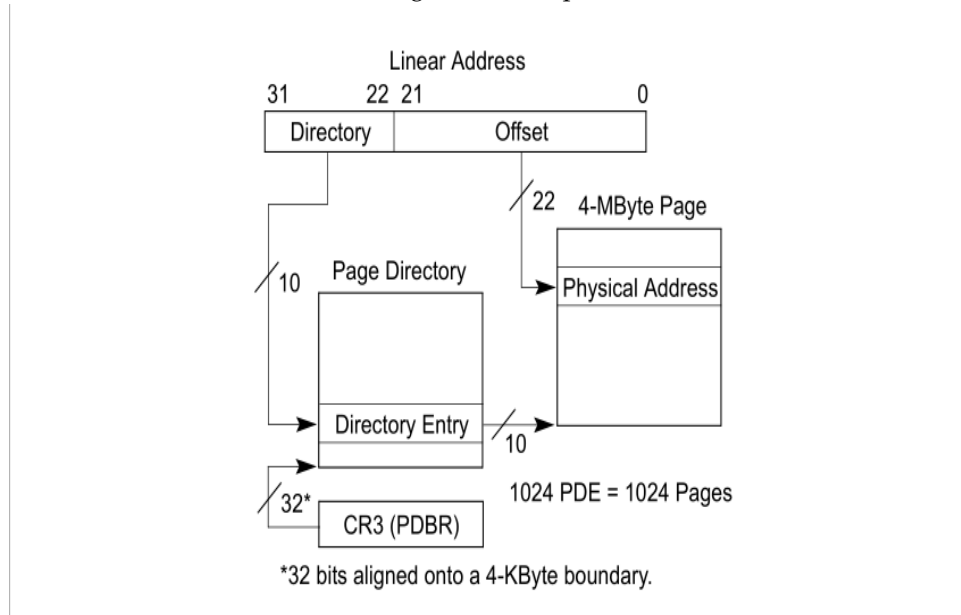
而在这之前, 之所kernel可以继续运行的原因就是因为它其实[10000,]与[KERNBASE.]对应同一物理地址, 因此在地址重定位之前, 可以正常的运行。

Challenge! We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE_PS ("Page Size") bit in the page directory entries. This bit was not supported in the original 80386, but is supported on more recent x86 processors. You will therefore have to refer to Volume 3 of the current Intel manuals. Make sure you design the kernel to use this optimization only on processors that support it!

首先, 查看intel手册3.6, 可以了解, 在开启了大页面后, 虚拟地址到物理地址的转变也会有原来的二级页表变为一页表, 通过简单的计算就可知其合理性。然后地址的转换过程请见Figure 4.

这里, 我们在重新开一下pde的格式 (Figure 5); 我们会发现, 原来被保留的两位其中的以为被用来控制是大页面或者小页面, 我们只要将其置为一, 就可以开启大页面了。

Figure 4: va to pa



事实上, 我们开启大页面之前, 需要判断一下机器支不支持大页面, 其判断方法如下:

```

1 #define PSE_REG(arg) __asm__ __volatile__ (\
2     "movl_$1, %%eax\n\t" \
3     "CPUID\n\t" \
4     "movl_$8, %%edi\n\t" \
5     "andl_%%edi, %%edx\n\t" \
6     : "=d" (arg) );

```

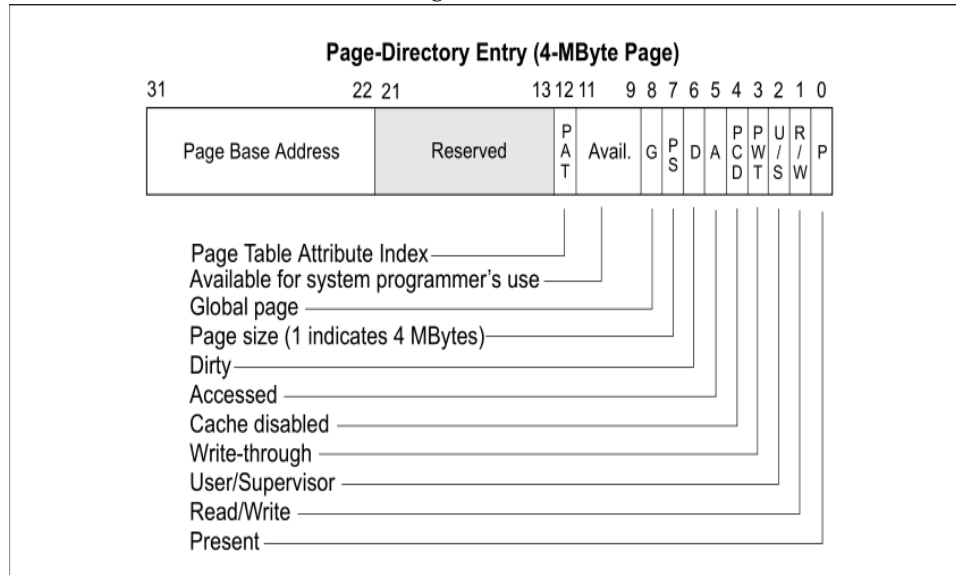
最后, 列出做过修改的代码如下(修改pgdir_walk())的代码比较简单, 只需加入对PTE.PS位的判断即可, 这里就不予以列出): kern/pmap.c

```

1 static void
2 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
3                 int perm)
4 {
5     // Fill this function in
6     if (!(perm & PTE_PS))
7     {
8         pte_t *tmp_pte;
9         int i;
10        pde_t *tmp_pde;
11        for (i=0; i<size/PGSIZE; i++)
12        {
13            //update pde
14            tmp_pde=(pde_t *) (pgdir+PDX(va));

```

Figure 5: PDE



```

14         tmp_pte=pgdir_walk(pgdir, (void *)va,1);
15
16         //update pte
17         *tmp_pte =0;
18         *tmp_pte |=perm|PTE_P;
19         *tmp_pte |= (physaddr_t) (pa+PGSIZE*i);
20         va+=PGSIZE;
21     }
22 }
23 else
24 {
25     pde_t *tmp_pde;
26     int i;
27     for(i=0;i<size/PTSIZE;i++)
28     {
29         tmp_pde =(pde_t *) (pgdir+PDX(va));
30         *tmp_pde |=perm;
31         *tmp_pde |= (physaddr_t) (pa+PTSIZE*i);
32         va +=PTSIZE;
33     }
34 }
35
36 }
```

kern/pmap.c meminit()

```

1     int pse;
2     PSE_REG(pse);
3     if(pse==PSE)
```

```

4      {
5          boot_map_region(kern_pgdir, KERNBASE, ~KERNBASE+1, PADDR
6              ((void *)KERNBASE), PTE_P | PTE_W | PTE_PS);
7
8          // Check that the initial page directory has been set up
9          // correctly.
10         check_kern_pgdir();
11         lcr4 (rcr4 () | CR4_PSE);
12     }
13     else
14     {
15         boot_map_region(kern_pgdir, KERNBASE, ~KERNBASE+1, PADDR
16             ((void *)KERNBASE), PTE_P | PTE_W);
17         check_kern_pgdir();
18     }
19 }

```

并且，利用challenge 2中的showmapping函数，可以看出，确实开启了大页面：

```

K> showmapping 0xf000000 0xf0800000
0xf000000 - 0xf03fffff  0x0 - 0x3fffff      kernel R/W user  -/-
0xf040000 - 0xf07fffff  0x400000 - 0x7fffff      kernel R/W user  -/-
K>

```

本实现机制只能实现对内核kernbase上的映射，用户地址空间部分并不能简单的开启大页面，因为讨论之后觉得这样做太复杂了，还需要解决如何在物理空间中找到一个连续的4M空间，因此没有给予实现。

Challenge! Extend the JOS kernel monitor with commands to:

Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter 'showmappings 0x3000 0x5000' to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000. Explicitly set, clear, or change the permissions of any mapping in the current address space. Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries! Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

第一个比较简单，仅需要根据给定的虚拟地址，找到对应的页目录即可。然后剩下的就是打印出页目录的信息，需要考虑的是给出的地址如果不是页大小的整数倍，这里我是将低地址向上去整，高地址向下取整（注意对0xffffxxx等高地址段的判断）。然后还需要加入对大页面的判断。然后直接输出就可以了，由于代码比较冗长，实现比较简单，这里就不贴出来了。运行结果如下：

```

K> showmapping 0xff000000 0xffffffff
0xff000000 - 0xff3fffff  0xf000000 - 0xf3fffff      kernel R/W user
-/-
0xff400000 - 0xff7fffff  0xf400000 - 0xf7fffff      kernel R/W user
-/-

```

```

0xff800000 - 0xffbfffff 0xf800000 - 0xfbfffff  kernel  R/W  user
-/-
0xffc00000 - 0xffffffff 0xfc00000 - 0xfffffff  kernel  R/W  user
-/-
K> showmapping 0xef000000 0xef006000
0xef000000 - 0xef000fff 0x11b000 - 0x11bfff  kernel  R/-  user  R/-
0xef001000 - 0xef001fff 0x11c000 - 0x11cfff  kernel  R/-  user  R/-
0xef002000 - 0xef002fff 0x11d000 - 0x11dfff  kernel  R/-  user  R/-
0xef003000 - 0xef003fff 0x11e000 - 0x11efff  kernel  R/-  user  R/-
0xef004000 - 0xef004fff 0x11f000 - 0x11ffff  kernel  R/-  user  R/-
0xef005000 - 0xef005fff 0x120000 - 0x120fff  kernel  R/-  user  R/-
K>

```

由上可以看出，运行结果是正确的。

第二个是可以任意的修改已经建立好的mapping的权限。可以设置，修改，清除标志位。这个功能也相对比较简单，首先是需要规定好格式的，我的定义如下：

map [op] [addr] [perm]

op的操作有三种，set，clear，change（其实个人认为set和change功能是相同的）。然后对于clear，不需要perm参数。我写出的代码如下：

```

1  int
2  mon_map(int argc, char **argv, struct Trapframe *tf)
3  {
4      if(argc>4 || argc<3)
5      {
6          cprintf("map_[op]_[addr]_[perm]_, less_than_4_args_are_
7              needed!\n");
8          return 0;
9      }
10     uintptr_t addr = strtol(argv[2], 0, 0);
11
12     pte_t *tmp_pte = pgdir_walk(kern_pgdir, (void *)addr, 0);
13     if(tmp_pte==NULL)
14     {
15         cprintf("There_is_no_pte_here!\n");
16         return 0;
17     }
18     if(strcmp(argv[1], "set")==0)
19     {
20         if(argc!=4)
21         {
22             cprintf("The_number_of_the_argvs_is_not_correct
23                 !\n");
24             return 0;
25         }
26         int perm = strtol(argv[3], 0, 0);
27         *tmp_pte &= 0xffff000;
28         *tmp_pte |= perm;
29     }
30     else if(strcmp(argv[1], "clear")==0)
31     {
32         *tmp_pte &= 0xffff000;
33     }
34 }

```

```

32     else if (strcmp(argv[1], "change")==0)
33     {
34         if (argc!=4)
35         {
36             cprintf("The_number_of_the_argvs_is_not_correct\n");
37             return 0;
38         }
39         int perm = strtol(argv[3], 0, 0);
40         *tmp_pte &= 0xffff000;
41         *tmp_pte |= perm;
42     }
43     else
44     {
45         cprintf("no_this_operation!\n");
46         cprintf("op_can_be: (1) set\n (2) change\n (3) clear\n");
47     }
48     return 0;
49 }

```

实现也比较简单，只是单纯的对权限为进行修改就可以了。做了一些测试，其结果如下：

```

K> showmapping 0xef001000 0xef003000
0xef001000 - 0xef001fff  0x11c000 - 0x11cfff      kernel  R/- user  R/+
0xef002000 - 0xef002fff  0x11d000 - 0x11dfff      kernel  R/- user  R/+
K> map clear 0xef001000
K> map change 0xef002000 2
K> showmapping 0xef001000 0xef003000
0xef001000 - 0xef001fff has not been mapped!
0xef002000 - 0xef002fff has not been mapped!
K> map set 0xef002000 3
K> showmapping 0xef001000 0xef003000
0xef001000 - 0xef001fff has not been mapped!
0xef002000 - 0xef001fff  0x11d000 - 0x11dfff      kernel  R/W user  -/+
K>

```

可以看出，三种职能都可以正常工作。

再看对内存内容的检查，通过物理地址和虚拟地址进行查看。需要考虑的事物理地址的情况，由于我们已经将0开始的物理地址与KERNBASE上的虚拟地址建立了映射，那么我们就可以利用提供的宏KADDR将物理地址转换为虚拟地址然后进行输出。代码如下：

```

1  int
2  pdump(physaddr_t addr, uint32_t size)
3  {
4      int i=0;
5      uint32_t c;
6      for(i=0; i<size*4; i+=4)
7      {
8          if((i%16==0) && (i!=0))
9          {
10             cprintf("\n");

```

```

11         cprintf("0x%x:~", addr+i);
12     }
13     else
14     {
15         if(i!=0)
16             cprintf("~");
17         else
18             cprintf("0x%x:~", addr+i);
19     }
20     if(addr+i>0xffffffff)
21     {
22         cprintf("\n");
23         return 0;
24     }
25     c=*(uint32_t *)KADDR(addr+i);
26     int a[8];
27     int j;
28     for(j=0; j<8; j++)
29     {
30         a[j]= ((c)>>(j*4)) & 0x0000000f;
31     }
32     cprintf("0x");
33     for(j=7; j>=0; j--)
34         cprintf("%x", a[j]);
35 }
36 cprintf("\n");
37 return 0;
38 }

```

虚拟地址的情况比较简单且实现基本相同，这里就不列举出来了。下边是对此功能的一些测试结果：

```

K> vdump 0xf000000 10
0xf0000000 : 0xf000ff53 0xf000ff53 0xf000e2c3 0xf000ff53
0xf0000010 : 0xf000ff53 0xf000ff53 0xf000ff53 0xf000ff53
0xf0000020 : 0xf000fea5 0xf000e987
K> pdump 0 10
0x0 : 0xf000ff53 0xf000ff53 0xf000e2c3 0xf000ff53
0x10 : 0xf000ff53 0xf000ff53 0xf000ff53 0xf000ff53
0x20 : 0xf000fea5 0xf000e987
K>

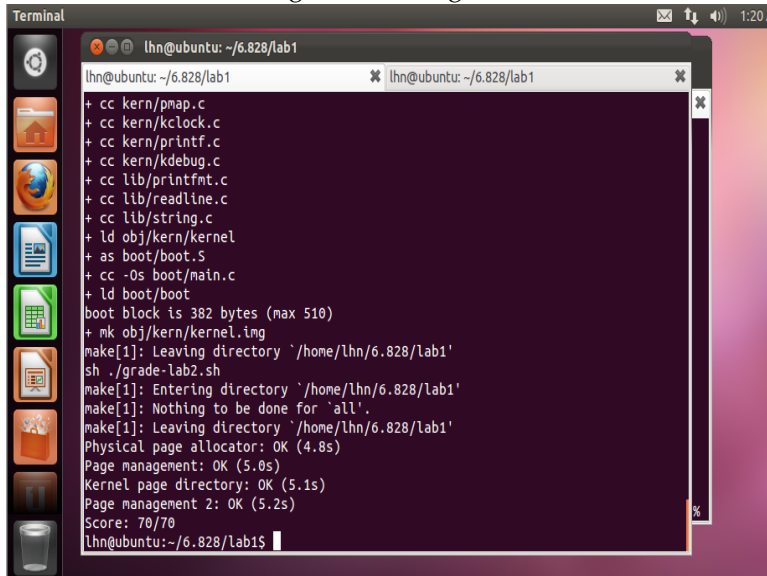
```

与Exercise3中的结果相同，结果正确。

3 RESULT

运行make grade，得到如下结果：见Figure6 make grade。

Figure 6: make grade



```
Terminal
lhn@ubuntu: ~/6.828/lab1
lhn@ubuntu: ~/6.828/lab1
+ cc kern/pmap.c
+ cc kern/kclock.c
+ cc kern/printk.c
+ cc kern/kdebug.c
+ cc lib/printk.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.o
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/lhn/6.828/lab1'
sh ./grade-lab2.sh
make[1]: Entering directory `/home/lhn/6.828/lab1'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/lhn/6.828/lab1'
Physical page allocator: OK (4.8s)
Page management: OK (5.0s)
Kernel page directory: OK (5.1s)
Page management 2: OK (5.2s)
Score: 70/70
lhn@ubuntu:~/6.828/lab1$
```

4 PROBLEM AND SOLUTION

1.起步阶段,在做第一个实验时,觉得无从下手,不明白整个实验到底是为了做些什么东西;然后,我就多向下看了很多实验,初步理解了后,在开始入手简单了许多。

2.关于boot.alloc()的最大内存问题,不能很确定,看了人人小组上的讨论后才豁然开朗。

3.关于pgdir_walk()函数的实现,关于pde权限的设定,开始没有理解的很透彻,结果后来的处理忘记了对它权限位进行设定,做到后边很远了,突然间冒出来莫名的错误,让我摸不到头脑。之后,我就再出错的函数中设置断点,查到了出错的地方,然后又认真的理解了pgdir_walk()的题意,在其中添加了pde权限位的设定,这是没问题的,因为x86中采用的是两级页表保护机制。

4.调用关系复杂,地址间的转换也比较普遍且繁多。在这个问题上,我出了很多的错误,后来就是重新的审视了一次,然后改正过来了。

5.在完成boot_map_region()对物理地址和虚拟地址的映射时,不是很清楚对于UPAGES位置的权限到底应该怎样设置。后来询问了同学,一起讨论后,解决了这个问题,报告中已经提及过了。

6.关于page_insert()最特殊情况(同一页面对同一虚拟地址重复进行映射时)的处理,开始也是利用特殊情况特殊判断的方法,后来仔细分析了题意后,发现,只要将ref的增加提前到page_remove()前即可,这样就不会导致释

放了不该释放的页面。

7.回答最后一个问题时,还是利用到了讨论的力量,因为obj/kern/kernel.asm中的地址和gdb调试时的地址不同,最后和同学讨论,觉得,反汇编的kernel.asm地址应该是根据VMA进行反汇编的,而kernel的VMA正好是0xf0100000。

5 THINKING AND HAVEST

1.合作交流还是必须的,前人经验还是有用的。

2.编程前应该提前考虑好所有的情况,弄清楚各个函数的调用关系,了解物理地址和虚拟地址的区别,看看什么时候用的物理地址,什么时候使用虚拟地址。我在实验过程中只是了解了大概就动手开始写代码了,结果总是出现一些莫名的错误。下次一定要注意。

3.对于边界情况的考虑欠缺,导致出现了很多莫名的错误,以后应该注意边界条件的考虑。

6 Suggestions

1.还是资料的提供方面,有一个challenge其实很难找到资料,如果能提供些相关的资料就好了。

2.人人网站是一个非常好的讨论版,上边的讨论解决了我很多问题。

7 Reference

书目:

《intel手册》

《深入理解计算机系统》

网站:

<http://grid.hust.edu.cn/zyshao/OSEngineering.htm>

大牛:

赵晓濛

何宇宁

刘洋