OS Lab2 实验报告

（郑懿 1611361）

一、 实验准备

本次试验在 Lab1 基础上进行操作，在了解了内存管理的机制后，对本次试验进行了多次尝试，并结合网上内容，最终完成实验。

二、 实验过程

练习 1:

在更改函数以前，我们先了解到，最重要的函数是 mem_init()，作为整个内存管理的初始化函数，在内核刚开始运行时就会调用。

进入该函数后，先检测现在系统可用空间，我们知道 JOS 把物理内存分为了三个部分：

一个是从 0x00000 ~ 0xA0000，这部分叫做 basemem，是可以使用的；

紧接着是 0xA0000 ~ 0x10000，这部分叫做 IO hole，是不可以使用的，用来分配给外部设备；

最后就是 0x10000，这部分叫做 extmem，是可以使用的，这是最重要的内存区域。

随后进行调用 boot_alloc()函数，并将返回值赋值给指向操作系统的页目录表的指针。

boot_alloc(uint32_t n):

对于该函数，通过阅读函数上方的注释，可以得知该函数只是在 JOS 设置其虚拟内存系统时用这个简单的物理内存分布器。已经给出了第一次调用该函数是语句，所以，只需要模仿，将存放着下一个可以使用的空闲内存空间的虚拟地址 nextfree 并在每次分配 n 字节的内存时候，更改这个变量的值。所以代码如下：

```
t_alloc(z)

    static char *nextfree;  // virtual address of next byte of free memory
    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) {
            char *next = nextfree;
            nextfree = ROUNDUP((char *) (nextfree+n), PGSIZE);
            return next;
    } else return nextfree;

    return NULL;
```

men_init(void):

随后，我们看下面的代码：

kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

这一条指令就是为页目录表添加第一个页目录表项，存放的就是操作系统的页表 kern_pgdir,，并通过 PADDR 计算 kern_pgdir 所对应的真实物理地址。

下一条代码，需要我们添加，这条命令要完成的是分配一块内存，用来存放一个数组，数组每一个 PageInfo 代表内存中的一页，并通过这个数组来追踪所有内存页的使用情况。代码如下：

```
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct PageInfo'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 PageInfo in this
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:
pages = (struct PageInfo *) boot_alloc(sizeof(struct PageInfo) * npages);
```

page_init(void):

随后，继续运行到下一个函数，page_init()，这个函数的功能由注释可知是初始化 pages 数组和 pages_free_list 链表，存放了所有空闲页的信息。并对 pages 初始化有了 4 个要求，为了实现这些要求，我们添加如下代码：

```
        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];
        }
        int med = (int)ROUNDUP(((char*)pages) + (sizeof(struct PageInfo) * npages) - 0xf0000000, PGSIZE)/PGSIZE;

        for (i = med; i < npages; i++) {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
}
```

因为 page 0 是被使用的，所以从 1 到 npages_basemem 进行初始化 ref 为 0，随后跳过 IO hole 部分，剩下的部分也进行初始化，所以分为两个循环进行初始化。

初始化关于所有物理内存页的相关数据结构后，进入 check_page_free_list(1)函数，该函数检查 page_free_list 空闲页是否合法、空闲。随后又通过 check_page_alloc(), 检查 page_alloc()，page_free()是否可以运行。

所以我们对这两个函数进行更改。

page_alloc(int alloc_flags):

关于该函数，注释告诉我们可以知道这个函数的功能是分配一个物理页，并返回这个物理页所对应的 PageIndo 结构体。

所以，该函数先从 free_page_list 中取出一个空闲页的 PageInfo 结构体，然后修改 free_page_list 的信息，然后取出空闲页 PageInfo 结构体信息，初始化该页的内存。代码如下：

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

//
```

*page_free(struct PageInfo *pp):*

根据注释，该函数就是把一个页的 PageInfo 结构体在返回给 page_free_list 空闲页链表，代表回收了这个页，通过完成以下几个步骤，修改回收的页 PageIndo 结构体的相应消息，把该结构体插入回 page_free_list 空闲链表中。代码如下：

```
void
page_free(struct PageInfo *pp)
{
        pp->pp_link = page_free_list;
        page_free_list = pp;
}
```

练习 2:

已经详细了解需要参考的知识。

练习 3:

```
(qemu) xp /10i 0x100000                  (gdb) x /10i 0xF00100000
0x00100000:  add    0x1bad(%eax),%dh        0x100000:    add    0x1bad(%eax),%dh
0x00100006:  add    %al,(%eax)              0x100006:    add    %al,(%eax)
0x00100008:  decb   0x52(%edi)              0x100008:    decb   0x52(%edi)
0x0010000b:  in     $0x66,%al               0x10000b:    in     $0x66,%al
0x0010000d:  movl   $0xb81234,0x472         0x10000d:    movl   $0xb81234,0x472
0x00100017:  push   %eax                    0x100017:    push   %eax
0x00100018:  adc    %eax,(%eax)             0x100018:    adc    %eax,(%eax)
0x0010001a:  mov    %eax,%cr3               0x10001a:    mov    %eax,%cr3
0x0010001d:  mov    %cr0,%eax               0x10001d:    mov    %cr0,%eax
0x00100020:  or     $0x80010001,%eax        0x100020:    or     $0x80010001,%eax
```

可以发现，两种方式并没有什么区别，只是了解虚拟地址和物理地址的区别。

问题 1:

假设以下 JOS 内核代码是正确的，变量 x 应该是什么类型？uintptr_t 还是 physaddr_t？

mystery_t x;

char * value = return_a_pointer（）；

* value = 10;

x =（mystery_t）value;

因为这里是用了*进行解析地址，所以 x 应该是 uintptr_t 类型。

练习 4:

pgdir_walk():

根据阅读注释，我们可知，该函数功能是：给定一个页目录指针 pgdir，该函数应该返回线性地址 va 所对应的页表项指针，并且根据参数 create 判断是否需要创建新的页表页。首先我们通过页目录表求的虚拟地址所在的页表页对于页目录项地址 dic_entry_ptr，随后判断这个页目录对应的页表页是否在内存中，有过在，那么计算其基地址 page_base，并且返回所对应页表项的地址&page_base[page_off]，如果不在，那么判断是否需要 create，如果需要，那么就把这个页的信息添加到页目录里。代码如下：

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        unsigned int page_off;
        pte_t * page_base = NULL;
        struct PageInfo* new_page = NULL;

        unsigned int dic_off = PDX(va);
        pde_t * dic_entry_ptr = pgdir + dic_off;

        if(!(*dic_entry_ptr & PTE_P))
        {
              if(create)
              {
                     new_page = page_alloc(1);
                     if(new_page == NULL) return NULL;
                     new_page->pp_ref++;
                     *dic_entry_ptr = (page2pa(new_page) | PTE_P | PTE_W | PTE_U);
              }
             else
                 return NULL;
        }

        page_off = PTX(va);
        page_base = KADDR(PTE_ADDR(*dic_entry_ptr));
        return &page_base[page_off];
}
```

boot_map_region():

该函数是为了把虚拟地址范围[va,va+size)映射到[pa,pa+size)的映射关系加入到页表中去，但只一部分的地址映射是静态的，并不会改变 pp_ref 的值，所以该函数只需要，在一个循环中，逐一进行映射即可，代码如下：

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    int nadd;
    pte_t *entry = NULL;
    for(nadd = 0; nadd < size; nadd += PGSIZE)
    {
        entry = pgdir_walk(pgdir,(void *)va, 1);    //Get the table entry of this page.
        *entry = (pa | perm | PTE_P);


        pa += PGSIZE;
        va += PGSIZE;

    }
}
```

page_lookup():

该函数的功能为：返回虚拟地址 va 所映射的物理页 PageInfo 结构体的指针，如果 pte_store 参数不为 0，则把这个物理页的页表项地址存放在 pte_store 中。我们只

需要调用 pgdir_walk 函数获取这个 va 对应的页表项，然后判断这个页是否已经存在内存，如果在则返回这个页的 PageInfo 结构体指针，并存放到 pte_store 中，代码如下：

```
//
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        pte_t *entry = NULL;
        struct PageInfo *result = NULL;

        entry = pgdir_walk(pgdir, va, 0);
        if(entry == NULL)
                return NULL;
        if(!(*entry & PTE_P))
                return NULL;

        result = pa2page(PTE_ADDR(*entry));
        if(pte_store != NULL)
        {
               *pte_store = entry;
        }
        return result;
}
```

page_remove():

该函数功能就是删除虚拟地址 va 和物理页映射关系，并且需要减少物理页上的引用计数，如果已经减少到 0，则应该是放物理页面没这个页对应的页表项应该被被置为 0.代码如下：

```
void
page_remove(pde_t *pgdir, void *va)
{
        pte_t *pte = NULL;
        struct PageInfo *page = page_lookup(pgdir, va, &pte);
        if(page == NULL) return ;
        page_decref(page);
        tlb_invalidate(pgdir, va);
        *pte = 0;
}
```

page_insert():

该函数的目的是将物理页面 pp 映射到虚拟地址 va，并且对于如果已有映射到 va 的页面，需要通过 page_remove()进行删除，插入成功，更改 pp_ref，如果页面以前为 va，则 TLB 必须无效，对与 TLB 相关，需要参考网上相关知识，并对随后的代码都有了帮助。

TLB：处理器使用 TLB（Translation Lookaside Buffer）来缓存线性地址到物理地址的映射关系。因此在世纪城的地址转换过程中，处理器首先更具线性地址查找 TLB，如果未发现该线性地址到物理地址的映射关系（TLB miss），将根据页表中的映射关系填充 TLB（TLB fill），然后进行地址转换。对于我们即将使用的 tlb_invalidate(pde_t *pgdir, void *va)函数，目的就是取消 va 与物理页之间的关联，相当于刷新 TLB，每次我们调整虚拟页和物理页之间映射关系的时候，都需要刷新 TLB，调用这个函数。

所以对于 insert 函数的实现，主要通过 pgdir_walk 函数求出虚拟地址 va 所对应的页表项，并且修改 pp_red 的值，通过查看这个页表项，确定 va 是否已经被映射，

如果被映射，则删除这个映射，把 va 和 pp 之间的映射关系加入到页表项，代码如下：

```
//
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
        pte_t *entry = NULL;
        entry =  pgdir_walk(pgdir, va, 1);
        if(entry == NULL) return -E_NO_MEM;

        pp->pp_ref++;
        if((*entry) & PTE_P)
        {
            tlb_invalidate(pgdir, va);
            page_remove(pgdir, va);
        }
        *entry = (page2pa(pp) | perm | PTE_P);
        pgdir[PDX(va)] |= perm;

        return 0;
}
```

练习 5:

映射 UPAGES，KSTACK，KERNBASE 等虚拟地址到物理内存。

mem_init():

有三个部分需要我们填充：

1、 要求是：Map 'pages' read-only by the user at linear address UPAGES，所以调用函数 boot_map_region 实现：

```
// 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);
```

2、 要求是：kernek RW，user NONE，所以代码如下：

```
// 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);
```

3、 要求是：kernel RW，user NONE，代码如下：

```
// 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:

boot_map_region(kern_pgdir,
        KERNBASE,
        -KERNBASE,
        0,
        PTE_W);
```

问题 2:

大致制作了一个表：

1023	0xffc00000	Page table for top 4MB of phys memory
...	...	...
960	0xf0000000	Page table for bottom of phys memory
959	0xefc00000	
958	0xef800000	ULIM
957	0xef400000	UVPT
956	0xef000000	UPAGES
955	0xeec00000	UPOP
...	...	..
0	0x00000000	Empty memory

问题 3：

因为页表可以设置权限，如果没有将 PTE_U 置为 1，就可以使用户无权限读写。

问题 4:

因为每个 PageInfo 占用 8Byte，而 UPAGES 最大是 4MB，所以总共最多可以有 4MNB/8B=512K 页，每页的容量是 4KB，所以总共可以有 512k*4KB=2GB。

问题 5:

如果有 2GB 内存，就需要有 512 个物理页，每个 PageInfo 结构占用 8Byte，所以是 4MB。页目录需要 512*8=4KB，此外还需要 512k 个页表项，所以还需要 4MB 存储，所以共消耗 6MB+4KB。

问题 6:

从 jmp *%eax 开始跳转，因为在 entry.S 中加载的事 entry_pgdir，他将虚拟地址[0,4M)和[KERNBASE,KERNBASE+4M)都映射到了物理地址[0,4M)，而在新的 kern_ogdir 加载后，并没有映射地位的虚拟地址[0,4M)，所以是必要的。

挑战 1:

上网寻找了一个解决方案是，通过注释掉先前的 KERNBASE 以上的 boot_map_region()，由于需要要使用 PDE 的 PS 位，所以通过 CR4 的 PSE 位，然后每个 PDE 表项对应 4MB 内存，所以就不需要分配二级页表原本 4KB 的页面大小，所以本来需要 64 个 PDE 表项和 64*1024 个 PTE 表项，也就是大约 256KB，现在 4MB 的页面大小，需要 64 个 PDE 表项，也就是 256B。代码如下：

```
uint32_t cr4;
cr4 = rcr4();
cr4 |= CR4_PSE;
lcr4(cr4);

// 设置 PDE
uintptr_t va = KERNBASE;
physaddr_t pa = 0;
size_t i;
for (i = 0; i < 64; ++i)
{
    kern_pgdir[PDX(va)] = pa | PTE_W | PTE_P | PTE_PS;
    va += PTSIZE;
    pa += PTSIZE;
}
```

并且注释掉 check_kern_pgdir()，防止检查二级目录，结果如下：

```
a(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef3ff]  PTE[000-3ff] -------U-P 00119-00518
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00118
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] --SDA---WP 00000 00400 00800 00c00 01000 01400 ..
  [ef7d1-ef7ff]  PTE[3d1-3ff] --S-----WP 04400 04800 04c00 05000 05400 05800 ..
[efc00-effff]  PDE[3bf]     -------UWP
  [efff8-effff]  PTE[3f8-3ff] --------WP 0010d-00114
[f0000-f43ff]  PDE[3c0-3d0] --SDA---WP 00000-043ff
[f4400-fffff]  PDE[3d1-3ff] --S-----WP 04400-0ffff
(qemu)
```

挑战 2:

参考网上代码，增添了 showmappings 的函数，在 kern/monitor 中增添指令，并且添加了如下代码：

```
int
mon_showmappings(int args, char **argv, struct Trapframe *tf)
{
    char flag[1 << 8] = {
        [0] = '-',
        [PTE_W] = 'W',
        [PTE_U] = 'U',
        [PTE_A] = 'A',
        [PTE_D] = 'D',
        [PTE_PS] = 'S'
    };

    char *arg1 = argv[1];
    char *arg2 = argv[2];
    char *arg3 = argv[3];
    char *endptr;
    if (arg1 == NULL || arg2 == NULL || arg3) {
        cprintf("we need exactly two arguments!\n");
        return 0;
    }
    uintptr_t va_l = strtol(arg1, &endptr, 16);
    if (*endptr) {
        cprintf("argument's format error!\n");
        return 0;
```

```
25.     }
26.     uintptr_t va_r = strtol(arg2, &endptr, 16);
27.     if (*endptr) {
28.         cprintf("argument's format error!\n");
29.         return 0;
30.     }
31.     if (va_l > va_r) {
32.         cprintf("the first argument should not larger than the second argume
    nt!\n");
33.         return 0;
34.     }
35.
36.     pde_t *pgdir = (pde_t *) PGADDR(PDX(UVPT), PDX(UVPT), 0);   // 这里直接
    用 kern_pgdir 也可以
37.     cprintf("      va range         entry      flag           pa range
    \n");
38.     cprintf("------------------------------------------------------------
    \n");
39.     while (va_l <= va_r) {
40.         pde_t pde = pgdir[PDX(va_l)];
41.         if (pde & PTE_P) {
42.             char bit_w = flag[pde & PTE_W];
43.             char bit_u = flag[pde & PTE_U];
44.             char bit_a = flag[pde & PTE_A];
45.             char bit_d = flag[pde & PTE_D];
46.             char bit_s = flag[pde & PTE_PS];
47.             pde = PTE_ADDR(pde);
48.             if (va_l < KERNBASE) {
49.                 cprintf("[%08x - %08x]", va_l, va_l + PTSIZE - 1);
50.                 cprintf(" PDE[%03x] --%c%c%c-
    -%c%cP\n", PDX(va_l), bit_s, bit_d, bit_a, bit_u, bit_w);
51.                 pte_t *pte = (pte_t *) (pde + KERNBASE);
52.         size_t i;
53.                 for (i = 0; i != 1024 && va_l <= va_r; va_l += PGSIZE, ++i)
    {
54.                     if (pte[i] & PTE_P) {
55.                         bit_w = flag[pte[i] & PTE_W];
56.                         bit_u = flag[pte[i] & PTE_U];
57.                         bit_a = flag[pte[i] & PTE_A];
58.                         bit_d = flag[pte[i] & PTE_D];
59.                         bit_s = flag[pte[i] & PTE_PS];
60.                         cprintf(" |-
    [%08x - %08x]", va_l, va_l + PGSIZE - 1);
```

```
                        cprintf(" PTE[%03x] --%c%c%c-
-%c%cP", i, bit_s, bit_d, bit_a, bit_u, bit_w);
                        cprintf(" [%08x - %08x]\n", PTE_ADDR(pte[i]), PTE_AD
DR(pte[i]) + PGSIZE - 1);
                    }
                }
                continue;
            }
            cprintf("[%08x - %08x]", va_l, va_l + PTSIZE - 1, PDX(va_l));
            cprintf(" PDE[%03x] --%c%c%c-
-%c%cP", PDX(va_l), bit_s, bit_d, bit_a, bit_u, bit_w);
            cprintf(" [%08x - %08x]\n", pde, pde + PTSIZE - 1);
            if (va_l == 0xffc00000) {
                break;
            }
        }
        va_l += PTSIZE;
    }
    return 0;
}
```

最后成功在 qemu 中调用了 showmappings 指令，结果如下：

```
He110 Worldx=3 y=-267296836K> showmappings
e need exactly two arguments!
> showmappings ef400000 f0000000
     va range         entry      flag           pa range
-------------------------------------------------------------------
ef400000 - ef7fffff] PDE[3bd] ----A--U-P
|-[ef7bc000 - ef7bcfff] PTE[3bc] -------UWP [003fd000 - 003fdfff]
|-[ef7bd000 - ef7bdfff] PTE[3bd] ----A--U-P [00119000 - 00119fff]
|-[ef7bf000 - ef7bffff] PTE[3bf] -------UWP [003fe000 - 003fefff]
|-[ef7c0000 - ef7c0fff] PTE[3c0] ----A--UWP [003ff000 - 003fffff]
|-[ef7c1000 - ef7c1fff] PTE[3c1] ----A--UWP [003fc000 - 003fcfff]
|-[ef7c2000 - ef7c2fff] PTE[3c2] ----A--UWP [003fb000 - 003fbfff]
|-[ef7c3000 - ef7c3fff] PTE[3c3] ----A--UWP [003fa000 - 003fafff]
|-[ef7c4000 - ef7c4fff] PTE[3c4] ----A--UWP [003f9000 - 003f9fff]
|-[ef7c5000 - ef7c5fff] PTE[3c5] ----A--UWP [003f8000 - 003f8fff]
|-[ef7c6000 - ef7c6fff] PTE[3c6] ----A--UWP [003f7000 - 003f7fff]
|-[ef7c7000 - ef7c7fff] PTE[3c7] ----A--UWP [003f6000 - 003f6fff]
```

三、 实验结果

最后通过 make grade 完成全部实验，结果如下：

```
running JOS: (0.6s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

通过本次试验加深了对内存管理的理解。也明白了，原来不能再 for 循环的要求里定义新的变量，而要先定义变量，再写循环。