Operating System MIT 6.828 JOS Lab2 Report

Computer Science
ChenHao(1100012776)

2013 年 10 月 3 日

目录

1 Part 1: Physical Page Management

1.1 Exercise 1

这个 Lab 意在让我们填充代码使得 JOS 能够正确使用新的的页寻址，由于具体寻址是由硬件实现的，对于内核的开发者需要做的只是做好内存的页的管理以及填写好初始 Page Directory 以及 Page Table 的值，之后将 cr3 指向我们新构建的表头，这样就可以使用新的页寻址了。

需要注意的是在这段代码开始前我们就已经开始使用了页寻址了，因此代码不可避免地需要进行虚拟地址的转化，因此在代码中要保持清醒的头脑什么时候需要使用物理地址什么时候需要使用虚拟地址。

1.1.1 boot_alloc()

直接 ROUNDUP 分配即可，注意要时刻保持 nextfree 对齐 PGSIZE。

end 是什么？end 是 linker 在链接的时候确定，其指向 kernel 的 bss 段的末尾。即 kernel 所使用的下一个空间的地址。end 是指向虚拟地址的指针。

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

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

    result = nextfree;
    nextfree = ROUNDUP(nextfree + n, PGSIZE);

    return result;
}
```

从 inc\memlayout.h 中可以知道 JOS 使用 struct PageInfo 以及 page_free_list 来管理每个 page，对于管理 pages 实际是一个链表，每个 page 保存其被引用的次数 (pp_ref) 以及在链表中下一个 free_page 的地址（pp_link，仅当当前页为 free_page 才有效），page_free_list 指向空页链表的头部。

1.1.2 mem_init() (only up to call to check_page_free_list(1))

需要建 npages 个 PageInfo 来管理 pages.

```
pages = (struct PageInfo *) boot_alloc(npages * sizeof(struct PageInfo));
```

1.1.3 page_init()

page_init 为初始化整个内存空间，并建立空闲页的链表数据结构。PageInfo 为页管理的数据结构。

需要知道哪些位置的物理内存是不能被使用的，注释中提到了第 0 页和 IO hole ([IOPHYSMEM, EXTPHYSMEM)) 是不能用来分配的，而对于 [EXTPHYSMEM, ...) 需要确定哪些是不可以使用的。EXTPHYSMEM 恰好就是 1MB 的位置，从 Lab1 中我们已经知道 Kernel 的代码和数据和栈都存放在了进接着 1MB 的位置, 而 extern char end[] 恰好是 kernel 存放的最后的地址（这里是虚拟地址，实际的物理地址为 end - KERNBASE）。而由 mem_init() 中可以知道，初始分配了两页的空间给 Initial Page Directory 和 PageInfo，这都是存放在紧接着 char* end 的连续的页中。因此第 0 页，以及 IOPHYSMEM 到 char* end 之后的已经使用的页均是不可分配的页，其余均是可行的。这里有个小的技巧，boot_alloc(0) 返回下一个未使用的内存的虚拟地址，因此可以直接计算得到不可分配页的最后的页编号。

```
void
page_init(void)
{
    page_free_list = NULL;
    size_t i;
    size_t nf_lb = IOPHYSMEM / PGSIZE;
    size_t nf_ub = PADDR(boot_alloc(0)) / PGSIZE;
    for (i = 0; i < npages; i++) {
        if (i != 0 && (i < nf_lb || i >= nf_ub)) {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        } else {
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
        }
    }
}
```

page_alloc 和 page_free 都只需要进行指针操作，调整 page_free_list 和其中表项的 pp_link 即可。

1.1.4 page_alloc()

```
// Hint: use page2kva and memset
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    while (page_free_list && page_free_list->pp_ref != 0)
        page_free_list = page_free_list->pp_link;
    if (page_free_list == NULL) {
        return NULL;
    } else {
        struct PageInfo * alloc_page = page_free_list;
```

```
        page_free_list = page_free_list->pp_link;
        if (alloc_flags & ALLOC_ZERO) {
            memset(page2kva(alloc_page), 0, PGSIZE);
        }
        return alloc_page;
    }
}
```

1.1.5 page_free()

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

2 Part 2: Virtual Memory

2.1 Exercise 2

Segment Translation translates logical address to linear address, then Page Translation translates linear address to physical address.

2.2 Exercise 3

How to access the QEMU monitor: Ctrl-a c

Some useful command:

xp/Nx paddr : display a hex dump of N words starting at physical address paddr.

info register : display all registers state, include GDT/LDT, seletor.

info mem : display mapped virtual memory and permissions.

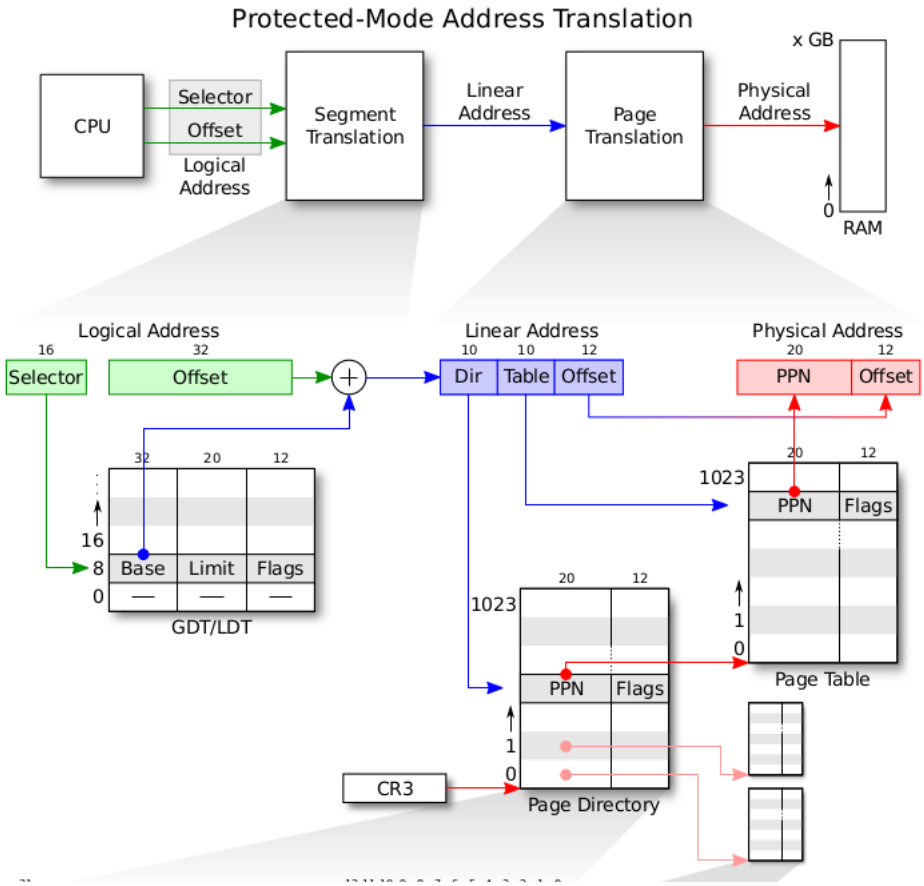
info pg : display the current page table structure.

2.3 Question

x 保存了 c 的指针，而 c 是指向虚拟内存地址的，因此 x 为 uintptr_t。

注意：由于操作系统也无法跳过虚拟内存转化的过程，因此无法通过物理内存直接访问，而需要通过转化为虚拟内存再进行访问，而 JOS 将实际地址 0x0 映射到 0xf0000000 处，因此对于操作系统内核来说物理地址于虚拟地址仅仅相差 0xf0000000。而 JOS 提供了 KADDR(pa) 来将物理地址转化为对应的虚拟地址，同时也提供了讲虚拟地址转化为物理地址的操作 PADDR(pa)。

2.4 Exercise 4



这个图片介绍了如何从虚拟地址转化到物理地址的，这个图片对我做这个 Lab 非常有帮助。

注意：在 JOS 中的 segment translation 在哪里？具体怎么实现的？如何使得 OS 能够正常运行？

这个 Exercise 4 一定要时刻保持清醒的头脑，在于 PAGE DIRECTORY,PAGE TABLE 中表项均是保存的是物理地址，而操作系统内核无法跳过虚拟内存转化过程，因此操作系统进行内存地址操作的时候一定要使用虚拟内存。

2.4.1 pddir_walk()

pddir_walk 将 vitural memory 对应的 page table entry 指针。如果不存在 PAGE TABLE，则创建一个 PAGE TABLE，并使得 PAGE DIRECTORY 对应项指向创建的 PAGE TABLE。

特别要注意 pgdir 中的所有项的中的 PPN 都是指的是物理地址，而内核无法避开使用虚拟内存转化的过程，因此在对指针进行取值操作的时候指针一定是要是指向虚拟地址的指针。这个地方我开始没考虑清楚，在该函数的 return 处的指针操作中没有加 KADDR() 而使用物理地址进行指针取值操作，导致疯狂 Triple Fault。

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // cprintf("pgdir_walk\n");
    if (pgdir[PDX(va)] == 0 || (pgdir[PDX(va)] & PTE_P) == 0) {
        // page table is not exist
        if (create == false) return NULL;

        struct PageInfo * new_page = page_alloc(1);
        if (new_page == NULL) return NULL;      // allocation fails
        ++new_page->pp_ref;
        pgdir[PDX(va)] = page2pa(new_page) | PTE_P | PTE_W | PTE_U;
    }
    return (pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])) + PTX(va);
}
```

2.4.2 boot_map_region

boot_map_region 将 [va, va+size) 虚拟地址映射到物理地址 [pa, pa+size) 上。对于虚拟地址映射的物理地址，只需要更改虚拟地址对应的 page table entry 的 PPN 值对应物理地址的高 20 位即可。因此可以使用 pgdir_walk 来进行取 (创建) 对应 page table entry。之后更改对应的 PPN 即可。

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm
    )
{
    // cprintf("boot_map_region\n");
    // size is a multiple of PGSIZE
    uintptr_t va_now;
    pte_t * pte;
    for (va_now = va; va_now != va + size; va_now += PGSIZE, pa += PGSIZE) {
        pte = pgdir_walk(pgdir, (void *)va_now, true);
        // 20 PPN, 12 flag
        *pte = pa | PTE_P | perm;
    }
}
```

注意：一开始我在 va_now != va + size 写成了 va_now < va + size，在 Part3 中一直 wa，之后才发现是这里写错了。我原本以为 uintptr_t 因为是 unsigned int 类型，所以在比较的时候会默认按照 unsigned 类型进行比较，不会造成问题。但是在 Part 3 的映射中，其中会使得 va + size == 2^{32}，而在 32 位机下就是等于 0，这样在使用 va_now < va + size 进行判断的时候就会恒为 false，造成错误。

2.4.3 page_lookup

返回虚拟地址 va 映射到的 Page，实际上就是 Page 的物理地址就是对应 page table entry 的 PPN。再利用 pa2page 转化到对应的 PageInfo 即可。

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // cprintf("page_lookup\n");
    // Fill this function in
    pte_t * pte = pgdir_walk(pgdir, va, 0);
    if (pte == NULL || (*pte & PTE_P) == 0) return NULL;   // no page mapped at
        va
    if (pte_store != 0) {
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

2.4.4 page_remove

删除虚拟地址 va 到物理地址的映射。需要做两部：第一删除对应 page table entry 的信息，第二删除 (减少引用次数) 映射的页。tlb_invalidate 是刷新 TLB 用，以防止因为 cache 导致错误的寻址。

```
void
page_remove(pde_t *pgdir, void *va)
{
    // cprintf("page_remove\n");
    // Fill this function in
    pte_t * pte;
    struct PageInfo * pg = page_lookup(pgdir, va, &pte);
    if (pg == NULL) return;
    page_decref(pg);
    if (pte != NULL) *pte = 0;
    tlb_invalidate(pgdir, va);
}
```

2.4.5 page_insert

将虚拟地址 va 映射到页 pp 上，如果原本有了别的映射，则需要删除原有映射。特别注意原有映射和 pp 相同的时候，因此方法很简单，先找到 va 对应的 page table entry，并先将 pp 的 pp_ref 增加，然后删去 va 原来对应的 page table entry 对应的映射，这里就是一个小技巧，由于 page_move 只是会减少引用次数，只有引用次数为 0 才会真正删除页。而我们先增加引用次数这样即使映射到相同的页也不会造成错误。

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // cprintf("page_insert\n");
    // Fill this function in
    pte_t * pte = pgdir_walk(pgdir, va, true);
    if (pte == NULL) return -E_NO_MEM;
    ++pp->pp_ref;
```

```
    if (*pte & PTE_P) {
        page_remove(pgdir, va);
    }
    *pte = page2pa(pp) | perm | PTE_P;
    return 0;
}
```

3 Part 3: Kernel Address Space

3.1 Exercise 5

在建立新的页寻址的时候，我们需要建立好新的映射。这样当我们将 cr3 指向新的 Page Directory 的时候就可以立即使用新的页寻址了。

按照 JOS 的要求：

1、[UPAGES, UPAGES + sizeof(PAGE)) 将会映射到 [pages, pages + sizeof(PAGE)) 以用于页面管理。

2、[KSTACKTOP-KSTKSIZE, KSTACKTOP) 将会映射到内核栈的实际物理地址，即在内核编译时.data 预留的一段区域，[bootstack, bootstacktop)。但是并不是所有都用于实际的栈中，栈有一部分将会作为 guard page，以防止内核栈溢出造成内核代码或数据的破坏，因此需要预留防止溢出的区域，并且这段区域不映射，当使用这个区域的时候将会造成 page fault，因此来保护操作系统内核。

3、[KERNBASE, 2^{32}) 将会映射到 [0, ...)。其中，这个映射直接保证了 JOS 之后启用新的页寻址，不会导致内核虚拟地址无法转化到实际的物理地址（内核的虚拟地址与实际地址仅相差 KERNBASE）。

3.1.1 map 'pages'

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

3.1.2 map 'bootstack'

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

3.1.3 map all of physical memory

```
    boot_map_region(kern_pgdir,
                    KERNBASE,
                    -KERNBASE,
                    0,
                    PTE_W);
  // in 32-bit system, 2^32 - KERNBASE = - KERNBASE
```

3.2 Question

3.2.1 Q2

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef020]  PTE[000-020] -------U-P 0011f-0013f
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 0011e
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] ----A--UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
  [ef7d1-ef7ff]  PTE[3d1-3ff] -------UWP 003ec 003eb 003ea 003e9 003e8 003e7 ..
[efc00-effff]  PDE[3bf]     -------UWP
  [efff8-effff]  PTE[3f8-3ff] --------WP 00113-0011a
[f0000-f03ff]  PDE[3c0]     ----A--UWP
  [f0000-f0000]  PTE[000]     --------WP 00000
  [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [f0100-f0104]  PTE[100-104] ----A---WP 00100-00104
  [f0105-f0119]  PTE[105-119] --------WP 00105-00119
  [f011a-f011a]  PTE[11a]     ---DA---WP 0011a
  [f011b-f011c]  PTE[11b-11c] --------WP 0011b-0011c
  [f011d-f011e]  PTE[11d-11e] ---DA---WP 0011d-0011e
  [f011f-f011f]  PTE[11f]     ----A---WP 0011f
  [f0120-f0120]  PTE[120]     ---DA---WP 00120
  [f0121-f013f]  PTE[121-13f] ----A---WP 00121-0013f
  [f0140-f03bd]  PTE[140-3bd] ---DA---WP 00140-003bd
  [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f3fff]  PDE[3c1-3cf] ----A--UWP
  [f0400-f3fff]  PTE[000-3ff] ---DA---WP 00400-03fff
[f4000-f43ff]  PDE[3d0]     ----A--UWP
  [f4000-f40fe]  PTE[000-0fe] ---DA---WP 04000-040fe
  [f40ff-f43ff]  PTE[0ff-3ff] --------WP 040ff-043ff
[f4400-fffff]  PDE[3d1-3ff] -------UWP
  [f4400-fffff]  PTE[000-3ff] --------WP 04400-0ffff
(qemu)
```

利用 qemu 的 info pg, 我们可以知道虚拟内存到物理内存的映射情况。

Base Virtual Address	Points to (logically):
ef000000-ef020000	pages
ef7bc000-ef7ff000	??
efff8000-effff000	Stack
f0000000-ffff000	KERN and Other

3.2.2 Q3

在 PDE 和 PTE 的 flags 位中，由 R/W 位，U/S 位控制是否可写，是否需要 supervisor 权限。这样即使是使用同样的地址空间可以避免用户使用错误的地址进行读写。

3.2.3 Q4

JOS 对于管理 PageInfo 的大小至多为 PTSIZE，在 mmu.h 中可知 PTSIZE=4M，而每个 PageInfo 需要 8-byte，因此至多有 512k 个页，因此最多支持 512k * 4M = 2G 的物理内存。

3.2.4 Q5

需要 1 页的 Page directory，Page directory 共 1024 项，每项需要一个页，因此需要 4M + 4K 来管理内存。由 Q4，PageInfo 的管理最多使用 PTSIZE，因此一共需要 8M+4K 来管理内存。

3.2.5 Q6

从 entrypgdir.c 中可以看出，页将 [0, 4MB) 和 [KERNBASE, KERNBASE + 4MB) 均映射到了物理地址的 [0, 4MB) 因此在开启页之后即使为 low eip，也能继续运行。知道 jmp 到高地才真正运行在 eip > KERNBASE 上。由于无法直接更改 eip，因此我们需要跳转语句来实现 eip 的执行到高位地址上，而在开启页到跳转期间依然是 low eip 执行，为了保证寻址正确，因此需要将虚拟地址的 [0, 4MB) 映射到物理地址的 [0, 4MB) 上。

3.3 Challenge 1:

PTE_PS 即使用大页直接映射 4MB 的空间，不需要二级页表，仅仅使用一级页表，这就减少了页表空间的开销，而且可以大大增加 TLB 的命中率。具体的映射方式在 Intel 手册上第三章有介绍，简单来说就是下面这张图：

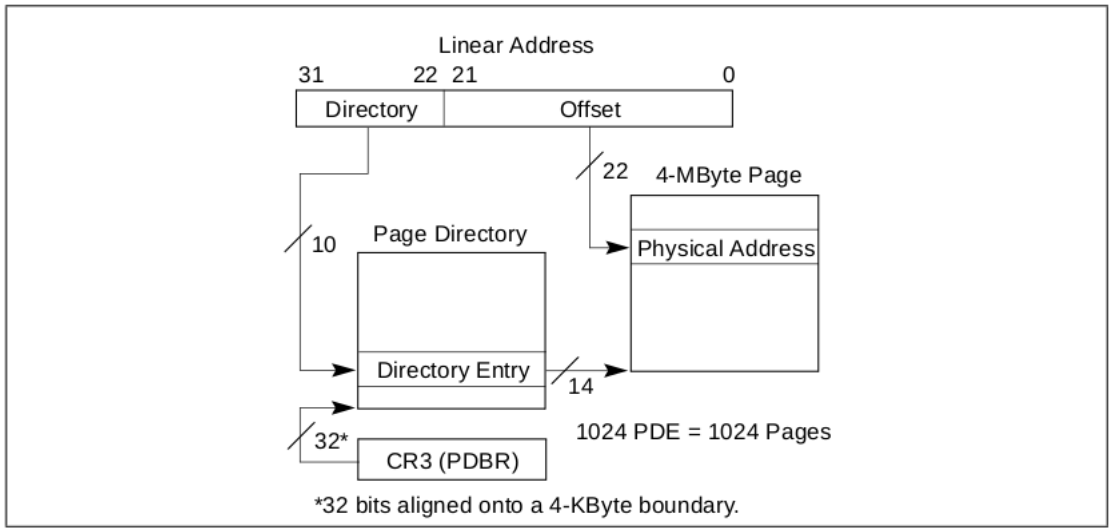


Figure 3-22. Linear Address Translation (4-MByte Pages)

:

实现的话应该类似二级页表，也是需要 PageInfo，以及预先构建映射表，处理好映射再开启直接使用即可。但是感觉实际需要我写可能会有点困难。。。

另外我觉得 4MB 的内存空间不太使用

3.4 Challenge 2:

3.4.1 ShowMappings

showmappings 为显示虚拟地址对应的物理地址。showmappings 只需要利用 pgdir_walk 到对应的 page table entry 即可。

```
int
mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 3) {
        cprintf("Command should be: showmappings [addr1] [addr2]\n");
        cprintf("Example: showmappings 0x3000 0x5000\n");
    } else {
        uint32_t laddr = strtol(argv[1], NULL, 0);
        uint32_t haddr = strtol(argv[2], NULL, 0);
        if (laddr > haddr) {
            haddr ^= laddr;
            laddr ^= haddr;
            haddr ^= haddr;
        }
        laddr = ROUNDDOWN(laddr, PGSIZE);
        haddr = ROUNDUP(haddr, PGSIZE);
        cprintf("0x%08x - 0x%08x\n", laddr, haddr);

        uint32_t now;
        pte_t *pte;
```

```
        for (now = laddr; now != haddr; now += PGSIZE) {
            cprintf("[ 0x%08x, 0x%08x ) -> ", now, now + PGSIZE);
            pte = pgdir_walk(kern_pgdir, (void *)now, 0);
            if (pte == 0 || (*pte & PTE_P) == 0) {
                cprintf(" no mapped \n");
            } else {
                cprintf("0x%08x ", PTE_ADDR(*pte));
                if (*pte & PTE_U) cprintf(" user       ");
                else cprintf(" supervisor ");
                if (*pte & PTE_W) cprintf(" RW ");
                else cprintf(" R ");
                cprintf("\n");
            }
        }
    }
    return 0;
}
```

```
K> showmappings
Command should be: showmappings [addr1] [addr2]
Example: showmappings 0x3000 0x5000
K> showmappings 0xf0000000 0xf0004000
0xf0000000 - 0xf0004000
[ 0xf0000000, 0xf0001000 ) -> 0x00000000  supervisor  RW
[ 0xf0001000, 0xf0002000 ) -> 0x00001000  supervisor  RW
[ 0xf0002000, 0xf0003000 ) -> 0x00002000  supervisor  RW
[ 0xf0003000, 0xf0004000 ) -> 0x00003000  supervisor  RW
K> showmappings 0x0 0x4000
0x00000000 - 0x00004000
[ 0x00000000, 0x00001000 ) ->  no mapped
[ 0x00001000, 0x00002000 ) ->  no mapped
[ 0x00002000, 0x00003000 ) ->  no mapped
[ 0x00003000, 0x00004000 ) ->  no mapped
K>
```

3.4.2 SetPermission

对于 set permission 类似直接通过 pgdir_walk 更改即可。

```
int
mon_setpermission(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 5) {
        cprintf("Command should be: setpermissions [virtual addr] [W (0/1)] [U
            (0/1)] [P (0/1)]\n");
        cprintf("Example: setpermissions 0x0 1 0 1\n");
    } else {
        uint32_t addr = strtol(argv[1], NULL, 0);
        uint32_t perm = 0;
        if (argv[2][0] == '1') perm |= PTE_W;
        if (argv[3][0] == '1') perm |= PTE_U;
        if (argv[4][0] == '1') perm |= PTE_P;
        addr = ROUNDUP(addr, PGSIZE);
        pte_t *pte = pgdir_walk(kern_pgdir, (void *)addr, 0);
        if (pte != NULL) {
            cprintf("0x%08x -> pa: 0x%08x\n old_perm: ", addr, PTE_ADDR(*pte));
            if (*pte & PTE_W) cprintf("RW"); else cprintf("R-");
```

```
            if (*pte & PTE_U) cprintf("U"); else cprintf("S");
            if (*pte & PTE_P) cprintf("P"); else cprintf("-");
            cprintf("  --> new_perm: ");
            *pte = PTE_ADDR(*pte) | perm;
            if (*pte & PTE_W) cprintf("RW"); else cprintf("R-");
            if (*pte & PTE_U) cprintf("U"); else cprintf("S");
            if (*pte & PTE_P) cprintf("P"); else cprintf("-");
            cprintf("\n");
        } else {
            cprintf(" no mapped \n");
        }
    }
    return 0;
}
```

```
K> setpermission 0xf0000000 0 0 1
0xf0000000 -> pa: 0x00000000
 old_perm: RWSP  --> new_perm: R-SP
K> showmappings 0xf0000000 0xf0001000
0xf0000000 - 0xf0001000
[ 0xf0000000, 0xf0001000 ) -> 0x00000000  supervisor  R
```

3.4.3 Dump

对于 dump，如果是查询虚拟内存的内容，则非常简单，直接输出即可。但是问题在于如何得到物理地址的内容，这对于已经建立好了页映射的操作系统来说是困难的，而对于 JOS，由于已经清楚地知道其物理内存和虚拟内存的映射关系，因此我根据映射关系来写就可以解决了。

pa_con(addr, *value) 为查找物理地址为 addr 的 4-byte 的值，返回是否存在值，如果存在则值保存在 value。

```
bool
pa_con(uint32_t addr, uint32_t * value)
{
    // get value in addr(physical address)
    // if no page mapped in addr, return false;
    if (addr >= PADDR(pages) && addr < PADDR(pages) + PTSIZE) {
        // PageInfo
        *value = *(uint32_t *)(UPAGES + (addr - PADDR(pages)));
        return true;
    }
    if (addr >= PADDR(bootstack) && addr < PADDR(bootstack) + KSTKSIZE) {
        // kernel stack
        *value = *(uint32_t *)(KSTACKTOP - KSTKSIZE + (addr - PADDR(bootstack)))
            ;
        return true;
    }
    if (addr < -KERNBASE) {
        // Other
        *value = *(uint32_t *)(addr + KERNBASE);
        return true;
    }
    // Not in virtual memory mapped.
    return false;
}
```

```
int
mon_dump(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 4) {
        cprintf("Command should be: dump [v/p] [addr1] [addr2]\n");
        cprintf("Example: dump v 0xf0000000 0xf0000010\n");
        cprintf("         dump contents in virtual address [0xf0000000, 0
            xf0000010)\n");
    } else {
        uint32_t laddr = strtol(argv[2], NULL, 0);
        uint32_t haddr = strtol(argv[3], NULL, 0);
        if (laddr > haddr) {
            haddr ^= laddr;
            laddr ^= haddr;
            haddr ^= laddr;
        }
        laddr = ROUNDDOWN(laddr, 4);
        haddr = ROUNDDOWN(haddr, 4);
        if (argv[1][0] == 'v') {
            // virtual address
            uint32_t now;
            pte_t * pte;
            for (now = laddr; now != haddr; now += 4) {
                if (now == laddr || ((now & 0xf) == 0)) {
                    if (now != laddr) cprintf("\n");
                    cprintf("0x%08x:  ", now);
                }
                pte = pgdir_walk(kern_pgdir, (void *)ROUNDDOWN(now, PGSIZE), 0);
                if (pte && (*pte & PTE_P))
                    cprintf("0x%08x  ", *((uint32_t *)now));
                else
                    cprintf("--------  ");
            }
            cprintf("\n");
        } else {
            // physical address
            uint32_t now, value;
            for (now = laddr; now != haddr; now += 4) {
                if (now == laddr || ((now & 0xf) == 0)) {
                    if (now != laddr) cprintf("\n");
                    cprintf("0x%08x:  ", now);
                }
                if (pa_con(now, &value)) {
                    cprintf("0x%08x  ", value);
                } else
                    cprintf("----------  ");
            }
            cprintf("\n");
        }
    }
    return 0;
}
```

```
K> dump v 0xf0000000 0xf0000010
0xf0000000:  0xf000ff53  0xf000ff53  0xf000e2c3  0xf000ff53
K> dump p 0x0 0x10
0x00000000:  0xf000ff53  0xf000ff53  0xf000e2c3  0xf000ff53
K> dump v 0xefff80000 0xefff80030
0xfff80000:  0x00000000  0x00000000  0x00000000  0x00000000
0xfff80010:  0x00000000  0x00000000  0x00000000  0x00000000
0xfff80020:  0x00000000  0x00000000  0x00000000  0x00000000
K>
```

3.5 Challenge 3:

暂时没查到相关资料，对于 kernel 在内核态和用户态的转化也不太清楚，等过段时间再回来做好了。。。

3.6 Challenge 4:

可以使用 Buddy Systems，基本思想是有大小为 2^m 的空间，每次请求某个大小都会分割离请求值向上取最近的 2 的次幂的大小。对于请求分配 2^k 大小的空间，首先会找到最小的空闲块，大小为 2^j，满足 $j >= k$，如果 $j = k$ 那么很好，则可以直接分配，否则我们递归对半分割 2^j 的空间，直到出现 2^k 大小的空间。对于清楚分配的空间也很简单，如果已经有相同大小的空闲块，则合并为一个更大的空闲块，直到不存在相同大小的空闲块。这些操作都可以利用二进制数进行位运算来进行，使得分配合并速度变快，缺点在于会有很多内部碎片。