

操作系统 第二次实验报告

周子锐 2100011032

2023 年 10 月 11 日

完成的 challenge 为:

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!)

Exercise 1

本题的要求为补全给出的 5 个函数.

`static void *boot_alloc(uint32_t n)`

在 `page_free_list` 建立前需要通过这个函数来分配内存, 根据需求模拟即可.

```
static void *
boot_alloc(uint32_t n)
{
    ...
}
```

```
// LAB 2: Your code here.

result = nextfree;
if (n > 0) {
    nextfree = (char *)ROUNDUP((uint32_t) nextfree + n, PGSIZE);
    if ((uint32_t)nextfree - KERNBASE > npages * PGSIZE)
        panic("boot_alloc: out of memory\n");
}
return result;
}
```

mem_init()

为 pages 申请空间并初始化为 0.

```
...
pages = (struct PageInfo *) boot_alloc(npages * sizeof(struct PageInfo));
memset(pages, 0, npages * sizeof(struct PageInfo));
...
```

void page_init(void)

根据注释将部分页面标为已使用, 这些页面不应该被加入到 page_free_list 中.

```
for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    if ((i == 0) || ((i >= IOPHYSMEM / PGSIZE) && i < kernel_end)) {
        pages[i].pp_link = NULL;
    } else {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

struct PageInfo *page_alloc(int alloc_flags)

分配一个物理页, 根据注释完成.

```
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    if (page_free_list == NULL) {
        return NULL;
    }
    struct PageInfo * nowpage = page_free_list;
    page_free_list = page_free_list -> pp_link;
    nowpage -> pp_link = NULL;
    if (alloc_flags & ALLOC_ZERO) {
        memset(page2kva(nowpage), 0, PGSIZE);
    }

    return nowpage;
}
```

void page_free(struct PageInfo *pp)

释放一个物理页, 也直接根据注释完成即可.

```
void
page_free(struct PageInfo *pp)
{
    // Fill this function in
    // Hint: You may want to panic if pp->pp_ref is nonzero or
    // pp->pp_link is not NULL.
    if ((pp->pp_link != NULL) || (pp->pp_ref != 0)) {
        panic("page_free: pp->pp_link != NULL or pp->pp_ref != 0");
    }
    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

Exercise 2

要求是阅读 Intel 80386 Reference Manual 的 5.2 节和 6.4 节.

5.2 节中描述了如何将一个 linear address 翻译成 physical address(在 lab2 中可以认为 linear address 和 virtual address 基本是一样的). 大致过程如下:

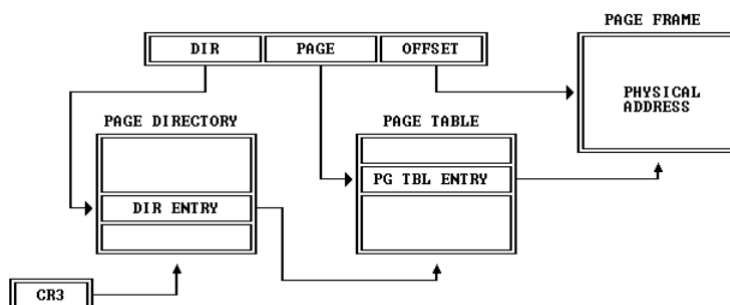
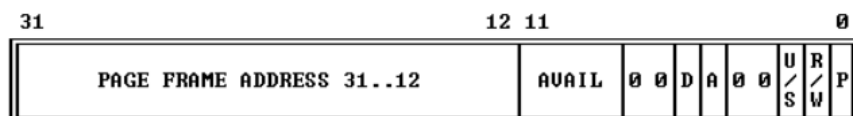


图 1: Page Translation Process

其中 page table entry 中的低 12 位被用来存储一些标志位, 具体意义如下



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.

图 2: Format of a Page-Table Entry

然后的 6.4 节就是对这些标志位进行说明:

- P-Present, 为 0 时这是一个无效的 PTE.
- A/D-Access/Dirty, 除去二级页表的 PTE 的 dirty 位外都由硬件设置; 在对页进行读或写操作前, 处理器将两级页表的 A 位都设置为 1; 在对 PTE 所覆盖的地址进行写操作前, 处理器将二级页表的 D 位设置为 1, 而一级页表的 D 位并没有定义.
- U/S-User/Supervisor, 该位与 CPL 有关, 当 CPL 为 0 或 1 或 2 时, 处于特权级, 此时可访问所有页; 当 CPL 为 3 时, 处于用户级, 此时只能访问 U/S 位为 1 的页.

- R/W-Read/Write, 为 0 时表示 read-only, 为 1 时表示 read-write. 处于特权级时所有页均可读写; 处于用户级时只能访问 R/W 位为 1 的页.

最后在 `inc/mmu.h` 中找到了如下定义:

```
// Page table/directory entry flags.
#define PTE_P 0x001 // Present
#define PTE_W 0x002 // Writeable
#define PTE_U 0x004 // User
#define PTE_PWT 0x008 // Write-Through
#define PTE_PCD 0x010 // Cache-Disable
#define PTE_A 0x020 // Accessed
#define PTE_D 0x040 // Dirty
#define PTE_PS 0x080 // Page Size
#define PTE_G 0x100 // Global
```

Exercise 3

映射建立后在 `qemu` 中查看 `0x00100000` 的内容 (这是内核被 `copy` 到的地址), 得到如下结果

```
(qemu) xp/4x 0x00100000
0000000000100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
```

图 3: `qemu` 中物理地址为 `0x00100000` 的内容

再在 `gdb` 中查看两处地址的内容:

```
(gdb) x/4x 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start-268435468>: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
```

图 4: 对应 `gdb` 中的内容

可以发现是完全一致的, 这也符合 `lab1` 中文档的描述.

Question 1

注意到这是一段 `kernel code`, 而 `kernel` 中只能使用虚拟地址访问内存, 从而 `x` 的类型为 `uintptr_t`.

Exercise 4

本题的要求依然是补全 5 个函数.

`pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create)`

这个函数在给定页目录 `pgdir` 下查找虚拟地址 `va` 对应的页表项, 同时如果此页表项没有被初始化且 `create` 为 1, 则创建一个新的页.

首先根据 `va` 的高 10 位计算得到 `dir entry` 的物理地址, 然后判断其 `P` 位的值来决定是否要创建新页, 最后根据 `dir entry` 的物理地址先转化为虚拟地址再加上 `va` 的中间 10 位来得到 `page table entry` 的地址.

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    pde_t *pde = pgdir + PDX(va);
    if (!((*pde) & PTE_P)) {
        if (create) {
            struct PageInfo *newpage = page_alloc(1);
            if (!newpage) {
                return NULL;
            }
            *pde = page2pa(newpage) | PTE_P | PTE_W | PTE_U;
            ++newpage->pp_ref;
        } else {
            return NULL;
        }
    }
    pte_t *pte = (pte_t *)KADDR(PTE_ADDR(*pde));
    pte += PTX(va);
    return pte;
}
```

```
static void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
```

这个函数在给定页目录 `pgdir` 下将虚拟地址 `va` 到 `va+size` 的区间映射到物理地址 `pa` 到 `pa+size` 的区间, 映射的权限为 `perm`.

暴力遍历每一对对应的 page, 通过 `pgdir_walk` 来得到 page table entry, 然后修改其值即可. 注意这里的映射不应该改变 `pp_ref`.

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

```
struct PageInfo * page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
```

这个函数在给定页目录 `pgdir` 下查找虚拟地址 `va` 对应的 `PageInfo`, 同时如果 `pte_store` 不为 `NULL`, 则将对应的 page table entry 的地址存入其中.

首先通过 `pgdir_walk` 得到 page table entry 的地址, 然后判断其可使用性, 最后将其转化为 `PageInfo` 的地址即可.

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if ((!pte) || (!(*pte & PTE_P))) {
        return NULL;
    }
    if (pte_store) {
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

page_remove(pde_t *pgdir, void *va)

这个函数在给定页目录 pgdir 下删除虚拟地址 va 对应的 PageInfo.

首先通过 page_lookup 得到对应的 PageInfo, 然后将其引用计数减一, 将对应的 page table entry 置零, 最后调用 tlb_invalidate 来使 TLB 失效.

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

page_insert()

这个函数在给定页目录 pgdir 下将虚拟地址 va 映射到 Page pp 所对应的物理地址, 映射的权限为 perm.

依旧是先找到对应的 page table entry, 然后先将 pp 的引用次数加一防止出现原来 va 就映射到 pp 所对应的 Page 的情况时会错误的减去其 pp_ref 的情况. 最后再写入新的 page table entry 即可.

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    pte_t* pte = pgdir_walk(pgdir, va, 1);
    if (!pte) {
```



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

Exercise 5

在 `inc/memlayout.h` 的注释中找到了详细的内存映射关系, 不在此赘述.

在 `kern/pmap.c/mem_init()` 中, 根据注释要求调用 `boot_map_region()` 函数完成映射.

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);
boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstacktop) - KSTKSIZE, PTE_W |
    PTE_P);
boot_map_region(kern_pgdir, KERNBASE, ((uint32_t)0xffffffff - KERNBASE), 0, PTE_W | PTE_P);
```

Question 2

从 `kern/pmap.c/mem_init()` 中按从上到下的顺序依次初始化了:

- UVPT 处初始化为 kernel page directory.
- UPAGES 处初始化为 pages 数组
- KSTACKTOP - KSTKSIZE 处初始化为 CPU0's Kernel Stack
- KERNBASE 初始化为之前所描述的高地址到低地址的映射.

进行一些计算后可以完成下表:

Entry	Base Virtual Address	Points to (logically):
1023		Page Table for top 4MB of phys memory
1022		
...		
960	0xf0000000	remapped physical memory(start form 0x00000000)
959	0xefc00000	CPU0's kernel stack and some invalid memory
957	0xef400000	kernel page directory
956	0xef000000	pages array
...		
0	0x00000000	[see next question]

Question 3

注意到 PTE 中有专门区别特权级和用户级的 U/S 位, 其中用户的程序只能读写 (或者只能读) 那些 U/S 位为 1 的页.

Question 4

在启动 qemu 时输出了如下的字样:

```
Physical memory: 131072K available, base = 640K, extended = 130432K
```

可以看到我们的内核使用了 128MB 的物理内存.

但实际上在 `kern/pmap.h` 中有如下的注释:

- This macro takes a kernel virtual address – an address that points above KERNBASE, where the machine's maximum 256MB of physical memory is mapped – and returns the corresponding physical address. It panics if you pass it a non-kernel virtual address.

因为在 JOS 中目前支持的地址映射就是将 KERNBASE 以上的虚拟地址映射到物理内存从 0 开始的位置, 也就是地址大于等于 0xf0000000 的部分, 一共是 256MB.

Question 5

此时我们认为 jos 能使用 256MB 的物理内存 (2^{16} 个 pages), 那么需要有

- 1 个 page directory, 占用 4KB;
- 至多 1024 个 page table, 占用 4MB;
- pages 数组, 占用 512KB.

从而需要的内存总量会略多于 4.5MB, 具体数值为 4612KB.

Question 6

在 kern/entry.S 中有如下的指令:

```
# Load the physical address of entry_pgdir into cr3. entry_pgdir
# is defined in entrypgdir.c.
movl $(RELOC(entry_pgdir)), %eax
movl %eax, %cr3
```

在 obj/kern/kernel.asm 中, 可以看到寄存器 %eax 此时被赋为了 0xf0116000, 从而能够跳转到 KERNBASE 之上.

从启动分页到执行此命令之前我们可以在低地址空间中执行是因为此时我们将虚拟地址的低地址和高地址部分都映射到了物理地址的 [0,4M) 中, 所以低地址和高地址访问到的内容是一致的.

这样的跳转是必要的是因为在这之后我们会建立新的映射关系, 而这样的映射关系缺乏从虚拟地址的低地址到对应的物理空间的地址的映射.

最终的评测结果如下:

```
+ ld boot/boot
boot block is 396 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/encodetalker/oslabs/lab'
running JOS: (0.9s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

图 5: 测试结果

Challenge

所有的代码均在 kern/monitor.c 中.

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.

这对应于 showmappings 命令, 直接遍历每一页即可.

```
int
mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 3) {
        cprintf("Usage: showmappings [start] [end]\n");
        return 0;
    }
    uintptr_t start_va = (uintptr_t)strtol(argv[1], NULL, 0);
    uintptr_t end_va = (uintptr_t)strtol(argv[2], NULL, 0);
    if ((start_va % PGSIZE) || (end_va % PGSIZE)) {
        cprintf("showmappings error: start and end must be page aligned\n");
        return 0;
    }
    if (start_va > end_va) {
        cprintf("showmappings error: start must be less than end\n");
    }
    while (start_va <= end_va) {
        pte_t *pte = pgdir_walk(kern_pgdir, (void *)start_va, 0);
        if ((!pte) || (!(*pte & PTE_P))) {
            cprintf("VA:0x%08x: unmapped\n", start_va);
        } else {
            cprintf("VA:0x%08x -> PA:0x%08x ", start_va, PTE_ADDR(*pte));
            if (*pte & PTE_U) {
                cputchar('U');
            } else {
                cputchar('-');
            }
            if (*pte & PTE_W) {
                cputchar('W');
            } else {
                cputchar(' ');
            }
        }
        start_va += PGSIZE;
    }
}
```

```

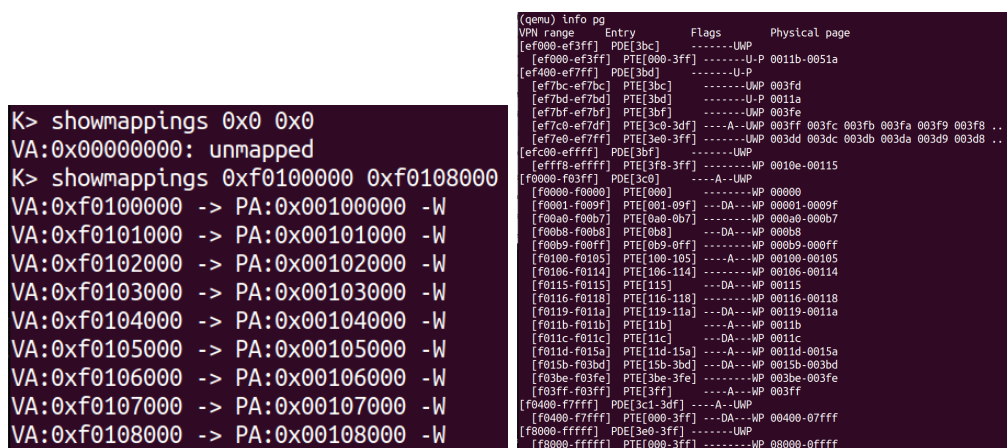
        cputchar('-');
    }
    cputchar('\n');
}

start_va += PGSIZE;
}

return 0;
}

```

运行示例如下:



```

K> showmappings 0x0 0x0
VA:0x00000000: unmapped
K> showmappings 0xf0100000 0xf0108000
VA:0xf0100000 -> PA:0x00100000 -W
VA:0xf0101000 -> PA:0x00101000 -W
VA:0xf0102000 -> PA:0x00102000 -W
VA:0xf0103000 -> PA:0x00103000 -W
VA:0xf0104000 -> PA:0x00104000 -W
VA:0xf0105000 -> PA:0x00105000 -W
VA:0xf0106000 -> PA:0x00106000 -W
VA:0xf0107000 -> PA:0x00107000 -W
VA:0xf0108000 -> PA:0x00108000 -W

(gemu) info pg
VPN range  Entry  Flags  Physical page
[ef000-ef3ff] PDE[3bc] -----U-MP
[ef000-ef3ff] PTE[000-3ff] -----U-P 0011b-0051a
[ef400-ef7ff] PDE[3bd] -----U-P
[ef7bc-ef7bc] PTE[3bc] -----U-MP 003fd
[ef7bd-ef7bd] PTE[3bd] -----U-P 0011a
[ef7bf-ef7bf] PTE[3bf] -----U-MP 003fe
[ef7c0-ef7d0] PTE[3c0-3df] -----A-MP 003ff 003fc 003fb 003fa 003f9 003f8 ..
[ef7e0-ef7ff] PTE[3e0-3ff] -----U-MP 003dd 003dc 003db 003da 003d9 003d8 ..
[efc00-effff] PDE[3bf] -----U-MP
[efc00-effff] PTE[3f0-3ff] -----W 0010e-00115
[f0000-f03ff] PDE[3c0] -----A-MP
[f0000-f0000] PTE[000] -----W 00000
[f0001-f009f] PTE[001-09f] ---DA--W 00001-0009f
[f00a0-f00b7] PTE[0a0-0b7] -----W 000a0-000b7
[f00b8-f00b8] PTE[0b8] ---DA--W 000b8
[f00b9-f00ff] PTE[0b9-0ff] -----W 000b9-000ff
[f0100-f0105] PTE[100-105] ---A--W 00100-00105
[f0106-f0114] PTE[106-114] -----W 00106-00114
[f0115-f0115] PTE[115] ---DA--W 00115
[f0116-f0118] PTE[116-118] -----W 00116-00118
[f0119-f011a] PTE[119-11a] ---DA--W 00119-0011a
[f011b-f011b] PTE[11b] ---A--W 0011b
[f011c-f011c] PTE[11c] ---DA--W 0011c
[f011d-f015a] PTE[11d-15a] ---A--W 0011d-0015a
[f015b-f03ae] PTE[15b-3ae] ---DA--W 0015b-003ae
[f03be-f03fe] PTE[3be-3fe] -----W 003be-003fe
[f03ff-f03ff] PTE[3ff] ---A--W 003ff
[f0400-f7fff] PDE[3c1-3df] -----A-MP
[f0400-f7fff] PTE[000-3ff] ---DA--W 00400-07fff
[f8000-fffff] PDE[3e0-3ff] -----U-MP
[f8000-fffff] PTE[000-3ff] -----W 00000-0ffff

```

图 6: showmappings 命令运行示例

Explicitly set, clear, or change the permissions of any mapping in the current address space.

对应 setperm 命令, 直接修改对应的 PTE 即可.

```

int
mon_setperm(int argc, char **argv, struct Trapframe *tf) {
    if (argc != 4) {
        cprintf("Usage: setperm [VADDR] [U|W] [0|1]\n");
        return 0;
    }

    if ((argv[2][0] != 'U' && argv[2][0] != 'W') || (argv[3][0] != '0' && argv[3][0] != '1')) {
        cprintf("Usage: setperm [VADDR] [U|W] [0|1]\n");
        return 0;
    }
}

```

```

}

uintptr_t va = (uintptr_t)strtol(argv[1], NULL, 0);
pte_t *pte = pgdir_walk(kern_pgdir, (void *)va, 0);
if (!pte) {
    cprintf("setperm error: VA:0x%08x is not mapped\n", va);
    return 0;
}

pte_t perm_mod = 0;
if (argv[2][0] == 'U') {
    perm_mod = PTE_U;
} else {
    perm_mod = PTE_W;
}

if (argv[3][0] == '1') {
    *pte |= perm_mod;
} else {
    *pte &= ~perm_mod;
}

return 0;
}

```

运行示例如下:

```

Type 'help' for a list of commands.
K> setperm 0xf0000000 W 0
K> QEMU 2.3.0 monitor - type 'help' for more information
(qemu) info pg
VPN range      Entry      Flags      Physical page
[ef000-ef3ff] PDE[3bc] -----U-MP
[ef000-ef3ff] PTE[000-3ff] -----U-P 0011b-0051a
[ef400-ef7ff] PDE[3bd] -----U-P
[ef7bc-ef7bc] PTE[3bc] -----U-MP 003fd
[ef7bd-ef7bd] PTE[3bd] -----U-P 0011a
[ef7bf-ef7bf] PTE[3bf] -----U-MP 003fe
[ef7c0-ef7df] PTE[3c0-3df] ----A--U-MP 003ff 003fc 003fb 003fa 003f9 003f8 ..
[ef7e0-ef7ff] PTE[3e0-3ff] -----U-MP 003dd 003dc 003db 003da 003d9 003d8 ..
[efc00-effff] PDE[3bf] -----U-MP
[efff8-fffff] PTE[3f8-3ff] -----W-P 0010e-00115
[f0000-f03ff] PDE[3c0] ----A--U-MP
[f0000-f0000] PTE[000] -----P 00000
[f0001-f009f] PTE[001-09f] ---DA---W-P 00001-0009f
[f00a0-f00b7] PTE[0a0-0b7] -----W-P 000a0-000b7
[f00b8-f00b8] PTE[0b8] ----DA---W-P 000b8
[f00b9-f00ff] PTE[0b9-0ff] -----W-P 000b9-000ff
[f0100-f0105] PTE[100-105] ----A--W-P 00100-00105
[f0106-f0114] PTE[106-114] -----W-P 00106-00114
[f0115-f0115] PTE[115] ----DA---W-P 00115
[f0116-f0118] PTE[116-118] -----W-P 00116-00118
[f0119-f011a] PTE[119-11a] ---DA---W-P 00119-0011a
[f011b-f011b] PTE[11b] ----A--W-P 0011b
[f011c-f011c] PTE[11c] ----DA---W-P 0011c
[f011d-f015a] PTE[11d-15a] ----A--W-P 0011d-0015a
[f015b-f03bd] PTE[15b-3bd] ---DA---W-P 0015b-003bd
[f03be-f03fe] PTE[3be-3fe] -----W-P 003be-003fe
[f03ff-f03ff] PTE[3ff] ----DA---W-P 003ff
[f0400-f7fff] PDE[3c1-3df] ----A--U-MP

```

图 7: setperm 命令运行示例

将其与 showmappings 的示意图进行对比, 可以看到确实成功修改了 0xf0000000 的标志位.

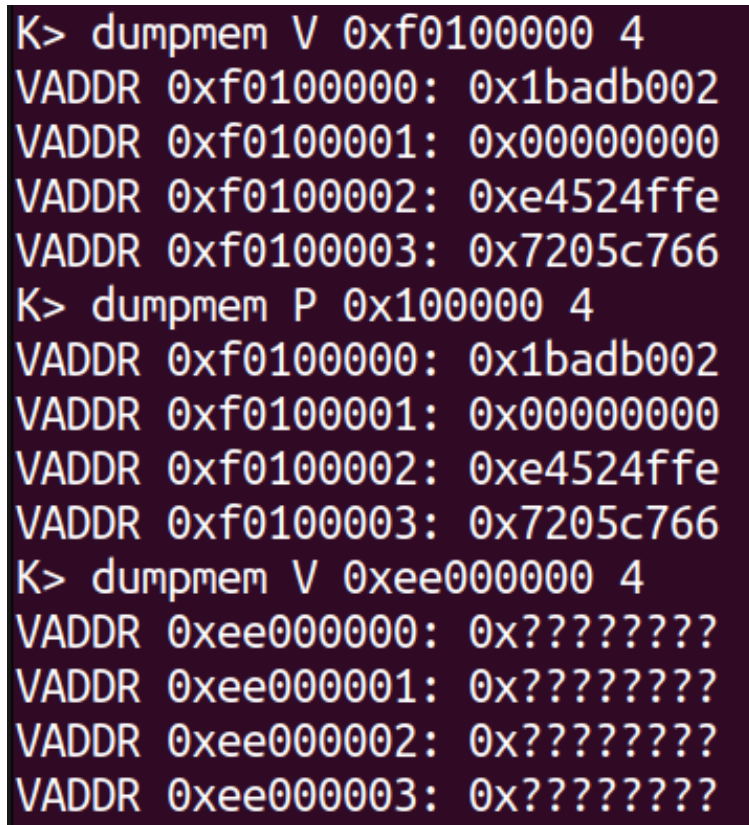
Dump the contents of a range of memory given either a virtual or physical address range.

对应 dumpmem 命令, 这里的实现是一个简化版本, 基于当前的 JOS 的若干特性. 详细细节见下, 其中对未 alloc 的部分输出为??.

```
int
mon_dumpmem(int argc, char **argv, struct Trapframe *tf) {
    if (argc != 4) {
        cprintf("Usage: dumpmem [V|P] [Start] [length]\n");
        return 0;
    }
    if ((argv[1][0] != 'V' && argv[1][0] != 'P')) {
        cprintf("Usage: dumpmem [V|P] [Start] [length]\n");
        return 0;
    }
    uintptr_t start_va = (uintptr_t)strtol(argv[2], NULL, 0);
    uint32_t length = (uint32_t)strtol(argv[3], NULL, 0);
    if (argv[1][0] == 'P') {
        if (start_va + length > PGSIZE * npages) {
            cprintf("dumpmem error: address overflow\n");
            return 0;
        }
        start_va = (uintptr_t)KADDR((physaddr_t)start_va);
    }
    for (int i = 0; i < length; i++) {
        cprintf("VADDR 0x%08x: 0x", start_va + i);
        for (int j = 3; j >= 0; j--) {
            void* nowptr = (void*)(start_va + i * 4);
            pte_t *pte = pgdir_walk(kern_pgdir, nowptr, 0);
            if ((!pte) || (!(*pte & PTE_P))) {
                cprintf("??");
            } else {
                cprintf("%02x", *((uint8_t *)nowptr + j));
            }
        }
        cprintf("\n");
    }
}
```

```
}  
    return 0;  
}
```

运行示例如下:



```
K> dumpmem V 0xf0100000 4  
VADDR 0xf0100000: 0x1badb002  
VADDR 0xf0100001: 0x00000000  
VADDR 0xf0100002: 0xe4524ffe  
VADDR 0xf0100003: 0x7205c766  
K> dumpmem P 0x100000 4  
VADDR 0xf0100000: 0x1badb002  
VADDR 0xf0100001: 0x00000000  
VADDR 0xf0100002: 0xe4524ffe  
VADDR 0xf0100003: 0x7205c766  
K> dumpmem V 0xee000000 4  
VADDR 0xee000000: 0x????????  
VADDR 0xee000001: 0x????????  
VADDR 0xee000002: 0x????????  
VADDR 0xee000003: 0x????????
```

图 8: dumpmem 命令运行示例