LAB2

516030910391 徐天强

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()，仅仅是在 JOS 建立自己虚拟内存系统时调用。之后分配页时，会调用 Page_alloc()。其中的 nextfree 指向下一个空闲的虚拟地址。

其中传入的参数 n，表示需要分配的 byte 数，由于 page 是 4K bytes align 的，因此需要做一个 ROUNDUP 操作。当 n=0 时，表示只是要查看一下当前下一个 free 的 byte 是哪一个，返回 nextfree 即可。当 n>0 时，将当前的 nextfree 作为 result 返回，并且后移一定的 PGSIZE 数即可。

page_init()，其功能是初始化 pages 包含的所有的 PageInfo 结构体。物理地址空间被我们划分为三部分。第一个 page，被认为是实模式下的 IDT 和 BIOS 结构体，它们是已经被分配的，因此 pp_ref=1.接下来的 160 个 page，被认为是 base memory，是空闲的。接下来到 boot_alloc 中的下一个 free memory 之前的内存都是被分配了的，所以 ref=1。之后的物理页被认为是 free 的。

分类讨论即可。

page_alloc()，首先检查是不是还有剩余的 free page，也即 page_free_list 是不是不为空，否则返回 NULL。当找到下一个 free page 的时候，将它从 page_free_list 中移除，并返回找到的 page。

page_free()，逻辑很简单，只要当当前的 page 的 ref 为 0 时，就将其插入 page_free_list 即可。注意，这里的逻辑只有这个操作，如果上层逻辑在错误的地方调用了这个函数，即 ppref!=0||pplink!=0 那么就调用 panic 报错。

Question

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

```
2.          mystery_t x;
3.          char* value = return_a_pointer();
```

```
4.          *value = 10;
```

```
            x = (mystery_t) value;
```

uintptr_t

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

```
        pgdir_walk()

        boot_map_region()

        boot_map_region_large() //  Map all phy-mem at KERNBASE as large
pages

        page_lookup()

        page_remove()

        page_insert()
```

1　pgdir_walk()，一共有三个参数。pdgir，指向 page directory table 的第一个 entry。va，即要找到匹配的 page 对应的 virtual address。Create，只有在出现 page fault 时候才起作用，也即当前的 page table 还没有被创建，如果 create=true，就创建一个，否则就返回 NULL。

fir_level 是第一级页表项，类型是 pde_t*. sec_level 是第二季页表项，类型是 pte_t*.

函数最后返回的是，指向这个 page 的第二级页表页表项的指针。

2　boot_map_region(),函数的作用是将[va, va+size]的虚拟地址映射到[pa, pa+size)。pdgir，指向 page directory table 的第一个 entry。第二个参数 va,是需要映射的虚拟地址的起始地址。Size 是要映射的大小。Pa 则是映射到的物理地址的起始地址。Perm 是需要添加的权限位。

每次使用 pgdir_walk，来找到 va 对应的 pte，然后往其中填入物理地址和权限位即可。

3　boot_map_region_large()，函数的作用是将[va, va+size]的虚拟地址映射到[pa, pa+size)，而每一个页是一个巨页，即 PTSIZE。

实现方式与 boot_map_region 相同，不过每次 va 和 pa 都增加一个 PTSIZE。

4　page_lookup()，返回的是映射到 va 的 page。

在函数中，我们使用 pgdir_walk 和 pa2page 函数。前者帮助我们找到对应的 PTE。后者帮助我们从物理地址转化成 PageInfo*的结构体。

5　page_remove()，解除 va 到物理 page 的映射。

借助 page_lookup()，我们找到 va 对应的 page。通过 page_decref，我们将对应的 page 的 ref 减一，在这个函数中，当 ref 等于 0 时，会帮住我们 free 这个 page。利用 pg_store，将对应的 PTE 的内容清空。因为我们将一个 entry 从页表移除了，也就是说页表被改变了，这个之后，可能 TLB 里有对这个 entry 的缓存，因此我们必须对这个页表进行一致性更新。

6 page_insert()，该函数建立起 PageInfo 到 va 的映射绑定。当我们发现已经有一个 page 对应了这个 va 时，利用 page_remove 来删除原来的映射，然后再插入新的映射。

Question2

2.

Entry		Base virtual address	Points to
1023		0xffc00000	Page table for top 4MB of phys memory
1022		0xff800000	
960		0xf0000000	Kernel stack top
959		0xefc00000	Part of them are Kernel stack
957		0xef400000	UVPT-kern_pgdir
956		0xef000000	Pages' physical address
2		0x00800000	
1		0x00400000	
0		0x00000000	

2. (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?

因为 kernel 管理这系统的资源，有着更高的权限，如果用户态可以读写内核态的内存，可能会引发安全性的问题。
我们在每个 PTE 中置上 U 位，即表示这个内存 page 只有特权用户才能读。这样就能保护内核态内存

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

2G，pages 的大小是 4MB，而一个 PageInfo 的结构体是 8Byte，那么就有 524288 个 entry，每个 entry 对应一个 4K 的 page，524288*4K=2G

2. 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?

4MB 的 pages 和 1 个 PD 和 1024 个 PT，就是 1025*4K+4MB=4K+8MB.
因为我们一定会那一部分的内存，去做页表和 pages，所以不可能用到全部的物理内存。

2. 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?

```
jmp *%eax

pde_t entry_pgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};
```

将[0, 4MB)和[KERNBASE, KERNBASE+4MB)都映射到[0, 4MB)的物理地址。

因为之后内核在高 2G 的虚拟地址运行，如果不做这个映射就不能继续执行程序。
因为要为用户程序腾出空间

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!)

我们首先在 commands 数组里注册好 showmappings 指令。

```
int mon_showmappings(int argc, char **argv, struct Trapframe *tf)
```

定义一个函数。

首先我们规定一定要有三个参数。

```
if(argc != 3){
        cprintf("usage:   %s   <start-virtual-address>   <end-virtual-
address>\n", __FUNCTION__);
        return 0;
}
```

接着，规定使用的数字必须是 16 进制。

```
if(argv[1][0]!='0'||argv[1][1]!='x'||argv[2][0]!='0'||argv[2][1]!='x'
){
        cprintf("the virtual-address should be 16-base\n");
        return 0;
}
```

使用 strtol 将参数转化为 32 位无符号数字。

并且加入 low_va 必须小于 high_va 的限制：

```
if(low_va > high_va){
    cprintf("the start-va should < the end-va\n");
    return 0;
}
```

接着我们开始 showmapping。当虚拟地址小于 KERNBASE 的时候，是 4K 的页，我们需要访问到 PTE 才能获得物理地址。当虚拟地址大于 KERNBASE 的时候，是 4M 的页，只需要访问 PDE 就能获得物理地址了。