JOS_Lab2_Report

姓名：方嘉聪， 学号：2200017849

I. Challenge

这里我只实现了前两个Challenge，即 Mixing 4MB & 4KB Page 和 Monitor Commands

1. 4MB Page

需要查询一下Intel IA32-3A 的手册，主要看了下 Section 3.6.1, 3.7.2 & 3.7.3。
启动 4MB 需要设置 `CR4` 中的 PSE(page size extensions) flag，此外为了区分使用 4KB 还是 4MB 需要设置一下 PTE 中的 PTE_PS 位。

- 在 `kern/pmap.c/mem_init()` 中在 将 `kern_pgdir` 载入 `cr3` 之前，开启 `cr4` 中的 PSE bit `lcr4(rcr4() | CR4_PSE)`。值得注意的是，不能在载入 `cr3` 后再开启，不然内核地址转换会出错。
- 按照手册中的建议Section 3.7.3

> A typical example of mixing 4-KByte and 4-MByte pages is to place the operating system or executive's kernel in a large page to reduce TLB misses and thus improve overall system performance.
>
> The processor maintains 4-MByte page entries and 4-KByte page entries in separate TLBs. So, placing often used code such as the kernel in a large page, frees up 4-KByte-page TLB entries for application programs and tasks.

我在这里将虚拟地址$[\text{KERNBASE}, 2^{32})$用大页映射到 $[0, 2^{32} - \text{KERNBASE})$ 的内核物理地址。为此类似 `boot_map_region()` 实现了一个函数 `large_boot_map_region()` 用于实现大页映射。具体实现上类似原函数（反而会更简单些，见下大页地址翻译方式）所以只需调用必要的宏并注意设置PTE_PS位即可 `pgdir[PDX(va + i)] = (pte_t)(pa + i) | perm | PTE_P | PTE_PS;`

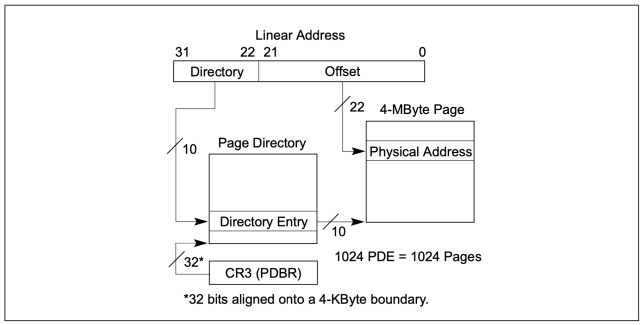

Figure 3-13. Linear Address Translation (4-MByte Pages)

- 为了顺利通过测试，还需要修改一下 `check_va2pa()` 函数，增加一下大页检测功能，只需判断是否为大页，若是，则提取 Page Dir Entry 中基地址再加上Offset即可 `return PTE_ADDR(*pgdir) | (PTX(va) << PTXSHIFT) | PGOFF(va);`
- 为了方便控制是否开启大页，我在 `kern/pmap.c` 中定义了一个控制的宏 `OPEN_LARGE_PAGE`，注释这个宏即可关闭大页（预计在后续lab里会关闭大页）。使用 `ifdef ...else... endif` 来控制上述大页的相关代码是否编译。
- 理论上还需要实现一个大页分配的函数以及修改 free_page_list 的具体实现来支持查询连续4MB的空闲页。（貌似是Challenge 5的内容，等有空闲时间再做吧）。

2. Monitor Commands

实现一些 memory management 相关的模拟器命令，按照要求一个个实现就行（是个体力活，与内核本身实现没什么关联），具体如下：

```
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Back trace the function call frames
showmappings - Display the physical page mappings that apply to [start_va, end_va]
setpermission - Set, clear or change the permissions of mappping at [va]
dumpmem - Dump the contents of a range of memory given either a virtual or physical addr.
```

1. `showmappings`：给定一个虚拟地址区间，输出映射的物理地址。实现的使用方法为 `Usage: showmappings <start_va> <end_va>`。大致思路是调用 `pgdir_walk()` 得到指向相应 PTE 的指针，判断是否存在映射并输出结果即可。效果如下：

```
K> showmappings 0xf0000000 0xf0010000
VisAddr: 0xf0000000, PhysAddr: 0x00000000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0001000, PhysAddr: 0x00001000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0002000, PhysAddr: 0x00002000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0003000, PhysAddr: 0x00003000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0004000, PhysAddr: 0x00004000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0005000, PhysAddr: 0x00005000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0006000, PhysAddr: 0x00006000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0007000, PhysAddr: 0x00007000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0008000, PhysAddr: 0x00008000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0009000, PhysAddr: 0x00009000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf000a000, PhysAddr: 0x0000a000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf000b000, PhysAddr: 0x0000b000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf000c000, PhysAddr: 0x0000c000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf000d000, PhysAddr: 0x0000d000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf000e000, PhysAddr: 0x0000e000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf000f000, PhysAddr: 0x0000f000, PTE_U: 0, PTE_W: 1
VisAddr: 0xf0010000, PhysAddr: 0x00010000, PTE_U: 0, PTE_W: 1
K> showmappings 0x00 0x00
VisAddr: 0x00000000, PhysAddr: no mappings
```

2. `setpermission`：设置给定虚拟地址映射的权限。使用方法为`Usage: setpermission [va] [U=0/U=1] [W=0/W=1]`，`U,W`分别表示 User 和 Write，也可以实现成`Usage: setpermission [va] [perm]`。类似上一个命令，判断是否存在映射，存在则更新权限即可，注意这里会将原有的权限清 0`*pte = (*pte & ~0xFFF) | perm | PTE_P;`，效果如下：

```
K> showmappings 0xf0000000 0xf0000000
VisAddr: 0xf0000000, PhysAddr: 0x00000000, PTE_U: 0, PTE_W: 1
K> setpermission 0xf0000000 U=1 W=0
K> showmappings 0xf0000000 0xf0000000
VisAddr: 0xf0000000, PhysAddr: 0x00000000, PTE_U: 1, PTE_W: 0
```

3. `dumpmem`：给一个（虚拟/物理）地址区间，十六进制输出存储的值。使用方法为`Usage: dumpmem [-p/-v] start_va length`。实现上比较的复杂，
   - 提供物理地址：注意到内核将所有的内核物理地址都线性映射到$[\mathrm{KERNBASE}, \ldots)$，调用提供的宏`KADDR`将物理地址转换到对应的内核虚拟地址，后输出对应内容即可。
   - 提供虚拟地址：调用`pgdir_walk()`讨论`page table page`不存在，PTE 不存在，PTE 存在三种情况，注意讨论跨页的情况

```
next = (uint32_t)PGADDR(PDX(start_addr) + 1, 0, 0) // Next PDE
next = (uint32_t)PGADDR(PDX(start_addr), PTX(start_addr) + 1, 0) // Next PTE
```

   - 效果如下:

```
K> dumpmem -v 0xf0000000 5
VisAddr: 0xf0000000, PhysAddr:0x00000000, 53
VisAddr: 0xf0000001, PhysAddr:0x00000000, ff
VisAddr: 0xf0000002, PhysAddr:0x00000000, 00
VisAddr: 0xf0000003, PhysAddr:0x00000000, f0
VisAddr: 0xf0000004, PhysAddr:0x00000000, 53
K> dumpmem -p 0x0 5
PA: 0x00000000, 53
PA: 0x00000001, ff
PA: 0x00000002, 00
PA: 0x00000003, f0
PA: 0x00000004, 53
PA: 0x00000005, ff
```

II. Exercise

Exercise 1:

需要实现`kern/pmap.c`中的`boot_alloc(), mem_init(), page_init(), page_alloc(),page_free()`几个页表物理地址初始化和分配的方法。核心在于维护`page_free_list`这个存储空闲物理页的链表。

1. `boot_alloc()`为 kernel 分配页，返回kernel的虚拟地址.
2. `mem_init()`调用相关函数初始化`napges, npages_basemem, kern_pddir`等等
3. `page_init(), page_alloc(),page_free()`实现页表初始化、分配和释放。对`page_free_list`这个链表进行操作就行。
4. **Debug:** 实现`page_init()`时，`[EXTPHYSMEM, kern_pgdir)`部分被 kernel 的`.bss`段使用，所以不能分配为 free。

Exercise 2 & 3:

复习页表转换和保护策略 (page-based protection) 熟悉 qemu monitor & gdb的相关用法.

- Question1: `mystery_t` 是 `uintptr_t`。由于 `value` 是指针，为虚拟地址。

Exercise 4:

需要实现 `kern/pmap.c` 中的 `pgdir_walk(), boot_map_region(), page_lookup(), page_remove(), page_insert().

1. `pgdir_walk()`: 输入虚拟地址和 page directory 基地址，返回指向对应 PTE 的指针。大致的实现思路是经由两级页表结构，结合下图的宏即可，注意判断是否对应的页表页是否存在。

```
A linear address 'la' has a three-part structure as follows:

+--------10------+-------10-------+---------12----------+
| Page Directory |   Page Table   | Offset within Page  |
|      Index     |      Index     |                     |
+----------------+----------------+---------------------+
 \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
 \---------- PGNUM(la) ----------/
```

2. `boot_map_region()`: 将物理地址 `[pa, pa+size)` 映射到虚拟地址 `[va, va+size)`。计算需要使用多少的页表，不断调用 `pgdir_walk()` 确认对应PTE，再将物理地址写入即可。
3. `page_lookup()`: 输入虚拟地址，返回对应的页，将PTE存储在 `pte_store` 中。注意如果不存在对应页，返回 `NULL` 即可。
4. `page_remove()`: 输入虚拟地址，将对应的物理页解除映射。按照提示调用 `page_lookup, tlb_invalidate, page_decref` 即可
5. `page_insert()`: 输入物理页 `pp` 和 虚拟地址 `va`，将 `va` 映射到 `pp`。按照注释中的提示一次实现即可，值得注意的是，针对 `va` 和 `pp` 实际对应相同的地址，那么需要在 `remove` 前，先将 `pp->pp_ref` 增加，以免出现在 `remove` 中因为 ref count 减小到0导致被释放的error。

```
void page_decref(struct PageInfo *pp)
{
    if (--pp->pp_ref == 0)
        page_free(pp);
}
```

6. Debug: 在 `pgdir_walk()` 的实现里，

```
        pte_t *pte = (pte_t *)KADDR(PTE_ADDR(*pgdir_entry)) + PTX(va);
```

注意要显式的指定类型转换 `pte_t *`，否则无法通过测试，发现不显式指定的话后续调用中得到的 pte 没法被正常修改指向的值。使用 `cprintf` 查看二者却没什么差异，不知道具体原因。。。

Exercise 5:

1. 实现 `mem_init()` 的剩余代码，认真分析 `memlayout.h`，反复调用 `boot_map_region` 即可，需要注意映射的 `size`，特别是2^{32} 使用 `1<<32` 会溢出，所以应当使用 `0xffffffff - KERNBASE + 1`。
2. Question 2:

Entry	Base Visual Address	Points to
1023	0xffc00000	Page table for top 4MB of phys memory
1022	0xff800000	Page table for second 4MB of phys memory
~	~	
~	~	
2	0x00800000	~
1	0x00400000	~
0	0x00000000	Page table for kernel i.e. the first 4MB of phys memory

3. Question 3: 主要是由于对于用户和内核使用的页，设置的 PTE_U 位不同，MMU会根据不同的CPL(Current Privilege Level)来设置。

4. **Question4:** 运行qemu输出的结果即为$131072\text{K} = 128\text{MB}$, 见下图

```
Booting from Hard Disk..
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> QEMU: Terminated
```

PS: 从理论上来分析，JOS使用二级页表，且PDE/PTE的大小都是一页，那么支持映射的最大虚拟内存范围是

$$1024 \times 1024 \times 4\text{KB} = 4\text{GB}$$

5. **Question 5:** 当完全映射时，虚拟页表(共两级)一共占用了 $(1024+1) \times 4\text{KB} \approx 4\text{MB}$。物理内存大小为$131072\text{K} \implies 32768 = 2^{15}\ \text{Pages}$，那么需要 $2^{15} \times 4\text{B} = 128\text{KB}$ 的物理页表项。此外JOS中还使用 `pages` 链表存储物理页状态。考虑到使用了分级页表，载入物理内存的虚拟页表会比较少，相对的实际开销会小于上述分析。
6. **Question 6:** 在指令

```
        mov     $relocated, %eax
        jmp     *%eax
```

后 `eip` 进入高地址。在开启分页(内存访问模式切换为虚拟内存)和 `eip` 转入高地址的过渡阶段，依靠 `kern/entrypgdir` 中的 `entry_pgdir()` 函数，将 `[KERNBASE, KERNBASE+4MB)` 和 `[0, 4MB)` 的虚拟地址都映射到物理地址 `[0, 4MB)`，使得内核可以正常运行。

III. Result

顺利通过所有测试

```
make[1]: Leaving directory '/home/ubuntu/6.828/lab'
running JOS: (0.5s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```