LAB 4 实验报告

姓名：杨智麟

学号：19300290039

任务一

`mmio_map_region`作用是将虚拟地址[base,base+size)映射到物理地址[pa,pa+size)上。其调用了`boot_map_region`来完成。

`boot_map_region`利用`pgdir_walk`来创建页表，然后将页表项中的地址设置为物理页的地址。

任务二

宏`MPBOOTPHYS`的作用是将虚拟地址映射到物理地址。

`mpentry.S`需要该宏是因为在链接时，使用物理地址；而在使用时，由`boot_aps`函数更改了位置：

```
static void
boot_aps(void)
{
    ...

    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);

    // Boot each AP one at a time
    ...
}
```

将位置改为了`MPENTRY_PADDR`作为入口地址，所以在使用时，需要将地址使用`MPBOOTPHYS`来映射保证正确。

而`boot/boot.S`不需要是因为其没有如上过程，链接和加载地址是相同的，我们使用`objdump`可以验证：

```
root@Ln-zlyang-server:/home/zlyang/OS/jos# objdump -h obj/boot/boot.out

obj/boot/boot.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000019c  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     0000009c  00007d9c  00007d9c  00000210  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00000870  00000000  00000000  000002ac  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      00000940  00000000  00000000  00000b1c  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      0000002a  00000000  00000000  0000145c  2**0
                  CONTENTS, READONLY
```

任务三

利用了宏`thiscpu`始终指向当前`CpuInfo`的特点在多cpu的环境下完成当前cpu的设定。

舍弃了全局`ts`变量而使用`thiscpu->ts`来进行代替。

通过`gdt[(GD_TSS0 >> 3) + i]`来设置cpu`i`的任务状态段的描述符。

任务四

q2

当不分开时，假设cpu0因中断而陷入内核，将信息压入栈中；此时，cpu2也因中断想要进入内核，虽然因为锁的缘故cpu2中断无法进入，但其仍然会将状态压入栈中，此时就导致cpu0的中断从内核退出时，从栈中弹出了错误的信息。

challenge

在`i386_init()`中，于BSP唤醒其他cpu之前加锁；

在`mp_main()`中，在初始化AP之前加锁；

在`trap()`中，于用户态切换时加锁；

在`env_run()`中，在切换回用户态之后释放锁。

任务五

q3

`Evn`数组`envs`被映射到虚拟地址`UENVS`，而用户环境的`env_pgdir`是基于`kern_pgdir`产生的，即对于`UTOP`上的地址映射关系在两个页表中是一样的。而`e`所对应的`Env`结构由操作系统管理，在虚拟空间地址都是`UENVS-UPAGES`的范围，因此在所有用户环境的映射也是一样的。

q4

用户环境进行环境切换是通过系统调用`syscall()`，最终通过`kern/trap.c`中的`trap()`产生异常然后陷入内核。

```
void trap(struct Trapframe *tf)
{
    ...

    if ((tf->tf_cs & 3) == 3)
    {
        // Trapped from user mode.
        // Acquire the big kernel lock before doing any
        // serious kernel work.
        // LAB 4: Your code here.
        assert(curenv);
        lock_kernel();

        // Garbage collect if current enviroment is a zombie
        if (curenv->env_status == ENV_DYING)
        {
            env_free(curenv);
            curenv = NULL;
            sched_yield();
        }
```

```
        // Copy trap frame (which is currently on the stack)
        // into 'curenv->env_tf', so that running the environment
        // will restart at the trap point.
        curenv->env_tf = *tf;
        // The trapframe on the stack should be ignored from here on.
        tf = &curenv->env_tf;
    }

    // Record that tf is the last real trapframe so
    // print_trapframe can print some additional information.
    last_tf = tf;

    // Dispatch based on what type of trap occurred
    trap_dispatch(tf);

    // If we made it to this point, then no other environment was
    // scheduled, so we should return to the current environment
    // if doing so makes sense.
    if (curenv && curenv->env_status == ENV_RUNNING)
        env_run(curenv);
    else
        sched_yield();
}
```

因而中断触发会进入`trapentry.S`的代码入口然后调用`trap()`，系统会在栈上创建一个`Trapframe`然后赋给用户环境的`env_tf`从而保护用户环境寄存器。具体便是`lab2`中的`_alltraps`部分：

```
_alltraps:
    ;ds es
    push %ds
    push %es
    pushal    #;其余寄存器

    #;load DS and ES with GD_KD (不能用立即数设置段寄存器)
    mov $GD_KD, %ax
    mov %ax, %ds
    mov %ax, %es
    pushl %esp
    call trap
```

这里压栈使得其结构与`Trapframe`一样，然后调用`trap()`就可以使得其作为`tf`被保存。