北航操作系统实验报告 Lab 3

学号：71066001

姓名：陈伟杰

思考题

> **Thinking 3.1** 请结合 MOS 中的页目录自映射应用解释代码中 `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V` 的含义。 ■

答：这行代码的含义是将当前环境的页目录（env_pgdir）中，与内核地址空间相关的页表项（UVPT 对应的页目录项所指向的页表）设置为自映射。总的来说，PDX(UVPT)是获取 UVPT 对应的页目录索引，然后将该索引处的页目录项设置为 PADDR(e->env_pgdir) | PTE_V。这里的 PADDR(e->env_pgdir)表示获取环境页目录的物理地址，然后进行按位或操作，将 PTE_V 位设置为 1，表示该页表项可用，通过这个操作，当前环境的页目录会创建一个自映射，将自己的页表映射到了虚拟地址 UVPT 处

> **Thinking 3.2** `elf_load_seg` 以函数指针的形式，接受外部自定义的回调函数 `map_page`。请你找到与之相关的 `data` 这一参数在此处的来源，并思考它的作用。没有这个参数可不可以？为什么？ ■

答：我们在 load_icode_mapper 被传入的 data 被用于这样一个语句中：

```
struct Env *env = (struct Env *)data;
```

data 参数的作用是将 load_icode_mapper 函数作为回调函数传递给 elf_load_seg 函数，在 elf_load_seg 函数中调用 load_icode_mapper 函数时，会将 struct Env 结构体的地址传递给 data 参数，以便 load_icode_mapper 函数能够获取到当前正在加载二进制代码的进程（即 env 变量）。因此，如果没有 data 参数，load_icode_mapper 函数就无法获取到当前进程的地址，总的来说没有进程指针，我们显然不能正常完成。

> **Thinking 3.3** 结合 `elf_load_seg` 的参数和实现，考虑该函数需要处理哪些页面加载的情况。 ■

答：该函数需要处理以下页面加载情况：

- 当 bin_size 大于等于 sgsize 时，直接将 ELF 文件中的二进制代码映射到虚拟地址空间中，不需要分配额外的页面。
- 当 bin_size 小于 sgsize 的时候，需要分配额外的页面，以便能够将 ELF 文件中的所有二进制代码都映射到虚拟地址空间中,如果此时还未达到 sgsize，则需要继续分配页面，将剩余的二进制代码映射到虚拟地址空间中，直到达到 sgsize 为止。

Thinking 3.4 思考上面这一段话，并根据自己在 **Lab2** 中的理解，回答：

- 你认为这里的 `env_tf.cp0_epc` 存储的是物理地址还是虚拟地址?

答：虚拟地址

Thinking 3.5 试找出上述 5 个异常处理函数的具体实现位置。

答：handle_int：kern/genex.S
handle_mod：kern/genex.S
handle_tlb：kern/genex.S
handle_tlb：kern/genex.S
handle_sys：kern/genex.S

Thinking 3.6 阅读 `init.c`、`kclock.S`、`env_asm.S` 和 `genex.S` 这几个文件，并尝试说出 `enable_irq` 和 `timer_irq` 中每行汇编代码的作用。

答：

71066001

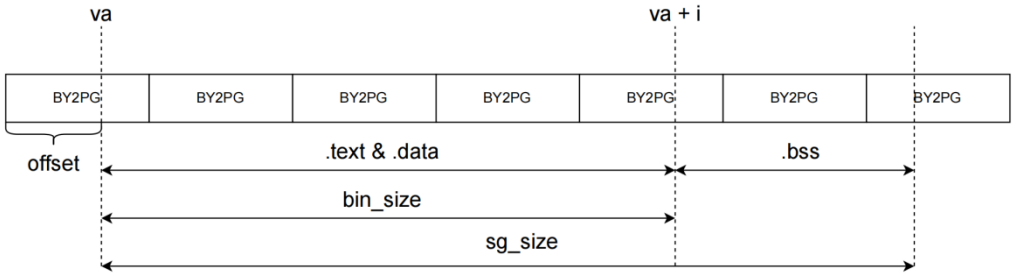
```
#include <asm/asm.h>
#include <mmu.h>
#include <trap.h>

.text
LEAF(env_pop_tf)
.set reorder
.set at
        sll     a1, a1, 6 //将 a1 寄存器中的值左移 6 位，相当于将其乘以 64，以得到物理地址。
        mtc0    a1, CP0_ENTRYHI//将物理地址存储到 CP0 寄存器 ENTRYHI 里，以便在异常处理过程中使用
        move    sp, a0//将 a0 寄存器中的值（即栈顶指针）存储到 sp 寄存器中，以便弹出 TrapFrame 结构体
        j       ret_from_exception//跳转到 ret_from_exception 函数，以便返回到之前的执行现场。
END(env_pop_tf)

LEAF(enable_irq)
        li      t0, (STATUS_CU0 | STATUS_IM4 | STATUS_IEc)//将 t0 寄存器设置为二进制 1001，即将 STATUS 寄存器的 CU0、IM4 和 IEc 位分别设置为 1。
        mtc0    t0, CP0_STATUS //将 t0 寄存器中的值存储到 CP0 寄存器 STATUS 中，以开启 CPU 中断
        jr      ra//跳转回 ra 寄存器中保存的返回地址，以便返回到调用该函数的位置
END(enable_irq)//函数将一个常量值赋给寄存器t0，然后将t0的值写入协处理器0的status寄存器中，使能中断，最后跳转到函数调用点。

```

```
71066001
END(handle_\exception)
.endm

.text

FEXPORT(ret_from_exception)
        RESTORE_SOME
        lw      k0, TF_EPC(sp)
        lw      sp, TF_REG29(sp) /* Deallocate stack */
.set noreorder
        jr      k0
        rfe
.set reorder

NESTED(handle_int, TF_SIZE, zero) //定义 handle_int 函数，并将其设置为中断处理函数，它将使用 TrapFrame 结构体作为参数，且不会使用任何寄存器
        mfc0    t0, CP0_CAUSE  //将 CP0 寄存器 CAUSE 中的值存储到 t0 寄存器中，以检测中断原因
        mfc0    t2, CP0_STATUS //将 CP0 寄存器 STATUS 中的值存储到 t2 寄存器中，以检测中断状态
        and     t0, t2 //将 t0 和 t2 寄存器中的值进行按位与操作，以得到中断原因和中断状态的交集
        andi    t1, t0, STATUS_IM4 //将 t0 和 STATUS_IM4 的值进行按位与操作，并将结果存储到 t1 寄存器中，以检测是否是计时器中断
        bnez    t1, timer_irq //如果 t1 寄存器中的值不为 0，说明是计时器中断，则跳转到 timer_irq 标签处
        // TODO: handle other irqs //如果不是计时器中断，则处理其他类型的中断
timer_irq:
        sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK) // 将值为 0 的字存储到设备 RTC 中断确认地址的内存位置中，以确认计时
器中断
        li      a0, 0 //将 a0 寄存器设置为 0，以表示计时器中断已经处理完毕
        j       schedule //跳转到 schedule 函数，以便进行任务调度
END(handle_int)
```

Thinking 3.7 阅读相关代码，思考操作系统是怎么根据时钟中断切换进程的。

答：操作系统首先会设置一个进程的队列然后会按照设定的时间间隔，向 CPU 发送时钟中断信号，当 CPU 接收到时钟中断信号时（判断中断信号就是我们设的那个时间间隔，一旦时间间隔走完），它就会停止当前正在执行的进程，并保存当前进程的上下文，接着，操作系统会从进程队列中选择下一个要执行的进程，并恢复该进程的上下文。CPU 接着就会开始执行下一个进程，

实现难点：在这个 lab 遇到的难点比如进程控制这一块没熟悉好，其中包括核心函数 env_alloc 的实现(通过 env_setup_vm 初始化虚拟内存(页表)，进程调度算法和加载二进制镜像 load_icode_mapper 函数,为什么需要这个函数？因为我们只需要提供给内核一个 ELF 文件，然后让内核将用户程序复制进去，就能够保证内核不变，用户随便。



//指导书给的关于 icode 需要分配的页面

笔者在这次 lab 的报错点主要是运行的时候是从低地址开始

```
init.c: mips_init() is called
Memory size: 65536K available, number of pages: 16384to memory 80430000 for struct Pages.
pmap.c:  mips vm init success
panic at tlbex.c:8 (passive_alloc): address too low
ra:    80102488  sp:  8013ff70  Status: 00080000
Cause: 00000008  EPC: 80102bc4  BadVA:  00000000
curenv:    NULL
cur_pgdir: 80400000
```

所以在找那个“明显”的 bug 可能是空指针或者野指针的时候发现,在 env_alloc 函数里发现我少了一个判断的返回

```
/* Step 1: Get a free Env from 'env_free_list' */
/* Exercise 3.4: Your code here. (1/4) */
e = LIST_FIRST(&env_free_list);
if (!e) {
        return -E_NO_FREE_ENV;
}
```

关于进程调度算法,我要判断如何调用这个函数，怎么使用静态变量来存储当前进程剩余执行次数，要了解整个进程调度就要知道如何来切换运行的进程的

体会感谢：在这次的 lab 学习到了很多，比如怎么创建一个进程并成功运行，怎么实现时钟中断，通过时钟中断内核可以再次获得执行权，学习了进程调度，在整个 lab 消耗的时间也非常多，在此感谢 os 课的助教答疑，让我迅速知道自己的错误并进行调试，整个 lab 的实验难度开始上升，自己每次做出来的时间也很久，很多 bug 其实都是因为自己没理解透彻所以才会有这个问题，希望自己继续努力，一步一步来，继续探索操作系统的乐趣！也希望继续学习到更多东西！