BUAA-OS Lab3 实验报告

姓名：苏云鹤
班级：212114
学号：21373007

1. 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。

根据页目录自映射的相关知识，假设页表基地址为 PT_{base}，
则页目录基地址为 $PD_{base} = PT_{base} + (PT_{base} >> 10)$，
自映射页目录项为
$PDE_{selfmap} = PT_{base} + PD_{base} >> 10 = PT_{base} + (PT_{base} >> 10) + (PT_{base} >> 20) = PD_{base} + (PT_{base} >> 20)$

由于 UVPT ~ ULIM 之间储存的是进程自己的页表，因此 UVPT 相当于 PD_{base}。PDX(UVPT)求的是UVPT的一级页表偏移，即 UVPT >> 22。因此， e->env_pgdir[PDX(UVPT)] = $PD_{base} + (PT_{base} >> 20)$。

在这个位置储存的应该是页目录的基地址PADDR(e->env_pgdir) | PTE_V。这也就是代码`e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V`的含义。

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？

`data`参数是一个指向env控制块的指针，因此在`elf_load_seg()`中其被转义：

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

而这个data是被这样传入的：

```
elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e)
```

e即为当前正在被操作的进程控制块指针。没有进程指针，我们的加载镜像的步骤无法正常实现。

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。

- `.text & .data`
  - 第一段可能有offset的偏移。由于页面的前半部分已经装载过内容，因此不能在此alloc或insert；
  - 中间段，正常处理；
  - 最后一段（该页的后半部分已经属于`.bss`段）。
- `.bss`
  - 第一段，前半页已经装载了`.text & .data`的相关信息；
  - 中间段，正常处理；
  - 最后一段，后半页不属于.bss。

Thinking 3.4.

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

是虚拟地址。

```
Elf32_Addr e_entry;       /* Entry point virtual address */
```

Thinking 3.5.

> 试找出 0、1、2、3 号异常处理函数的具体实现位置。8 号异常（系统调用） 涉及的 do_syscall() 函数将在 Lab4 中实现。

1. handle_int在kern/genex.S中实现。

```
NESTED(handle_int, TF_SIZE, zero)
    mfc0    t0, CP0_CAUSE
    mfc0    t2, CP0_STATUS
    and     t0, t2
    andi    t1, t0, STATUS_IM4
    bnez    t1, timer_irq
    // TODO: handle other irqs
timer_irq:
    sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK)
    li      a0, 0
    j       schedule
END(handle_int)
```

2. handle_mod引用的函数 do_tlb_mod定义在 kern/tlbex.c中
3. handle_tlb引用的函数do_tlb_refill定义在 kern/tlb_asm.S中
4. handle_sys引用的函数do_syscall是系统调用函数。

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。

```
# 开启中断
LEAF(enable_irq)
    li      t0, (STATUS_CU0 | STATUS_IM4 | STATUS_IEc)  # 将 t0 赋值成 (STATUS_CU0 |
STATUS_IM4 | STATUS_IEc)
    # CU0：“协同处理器0可用”：能够在用户模式下使用一些名义上的特权指令
    # IM4：4号中断可以被响应
    # IEc：开启中断
    mtc0    t0, CP0_STATUS  # 并用CP0_STATUS记录该状态
    jr      ra  # 返回
END(enable_irq)
```

```
timer_irq:
    sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK)
    # 在地址0xb5000110写0以响应时钟中断
    li      a0, 0
```

```
    # 将第一个参数设置为0
    j       schedule
    # 跳转至schedule函数
```

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。

操作系统中设置了一个进程就绪队列，用于管理所有处于就绪状态的进程。同时，为每个进程设置一个时间片，用于计时。每当正在运行的时间片走完，则需要执行时钟中断操作。这时，就将这个进程移动到就绪队列得尾端，并恢复其时间片；再让就绪队列最前端的进程执行。

---

2. 难点分析

2.1. 寄存器赋值

```
e->env_tf.cp0_status = STATUS_IM4 | STATUS_KUp | STATUS_IEp;
```

将 `status` 寄存器的值设置为 `STATUS_IM4 | STATUS_KUp | STATUS_IEp`，表示响应 4 号中断，是用户状态且开启中断。所有的通用寄存器状态在 `Trapframe` 中存储在 `regs` 数组中，其中第 29 号寄存器为 sp 寄存器。

2.2. 时钟中断过程

- 发生异常
- 处理器进入异常分发程序（本试验中是`exc_gen_entry`函数，位于`kern/entry.S`）,从异常向量组 `exception_handlers`，定位对应异常处理函数。
- 中断处理程序`handle_int`判断 `Cause` 寄存器是不是对应的 4 号中断位引发的中断，如果是，则执行中断服务函数 `timer_irq`。
- 中断服务函数 `timer_irq` 响应时钟中断，并进入调度程序 `schedule` 调度新进程。
- 调度程序 `schedule` 将时间片用尽的进程换为新进程，调用进程切换函数 `env_run` 使其运行。
- 进程切换函数 `env_run` 将原进程上下文保存，将新进程寄存器状态写入cpu中，实现进程切换。

---

3. 实验体会

1. 初步掌握了操作系统管理进程的流程；
2. 初步掌握了操作系统处理中断过程的细节；
3. 进一步体会到了os实验的魅力~